AFRL-RI-RS-TR-2008-61
Final Technical Report
March 2008

GEMINI: EXTENDING INFORMATION MANAGEMENT FOR REAL TIME TACTICAL ENVIRONMENTS

General Dynamics C4 Systems, Inc.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2008-61 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
JAMES HANNA Work Unit Manager	JAMES W. CUSACK Chief, Information Systems Division Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAR 2008	Final	Apr 06 – Feb 08

4. TITLE AND SUBTITLE

GEMINI: EXTENDING INFORMATION MANAGEMENT FOR REAL TIME TACTICAL ENVIRONMENTS

5a. CONTRACT NUMBER
FA8750-06-C-0067

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
63789F

6. AUTHOR(S)

Derek Merrill

5d. PROJECT NUMBER
C41E

5e. TASK NUMBER
06

5f. WORK UNIT NUMBER
02

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
General Dynamics C4 Systems, Inc.
8201 E McDowell Rd
Scottsdale AZ 85257-3812

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RISE
525 Brooks Rd
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2008-61

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# WPAFB 08-0445

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This is the Final Technical Report for the Joint Battlespace Infosphere (JBI) Gemini program, herein referred to as Gemini. The objective of this document is to describe the technical lessons learned, results achieved, and characteristics of the product offering which we have developed within this initial Gemini contract. Gemini focused on the challenges presented by highly dynamic networks and implementing a relatively lightweight and yet robust architecture. JBI is an information management system that provides users with the specific information required for them to perform their functional responsibilities during crisis or conflict. JBI integrates data from a wide variety of sources, aggregates this information, and distributes the information in the appropriate form and level of detail to users at all echelons. The "JBI platform" for brokering information from a distributed "JBI repository" provides the following core services to "JBI clients": publish, subscribe, query, and control.

15. SUBJECT TERMS
Information management, Net-centric, Communities of Interest, Unmanned Air Systems

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			James Hanna
U	U	U	UL	63	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

Table of Figures

Table of Tables

1 Scope

1.1 Identification and Objective

This is the Final Technical Report for the Joint Battlespace Infosphere (JBI) Gemini program, herein referred to as Gemini.

The objective of this document is to describe the technical lessons learned, results achieved, and characteristics of the product offering which we have developed within this initial Gemini contract.

1.2 Background

The Gemini goal is to design and build an innovative peer-to-peer information management software infrastructure for manned and unmanned tactical assets by leveraging previous JBI implementation experience.

JBI is an information management system that provides users with the specific information required for them to perform their functional responsibilities during crisis or conflict. JBI integrates data from a wide variety of sources, aggregates this information, and distributes the information in the appropriate form and level of detail to users at all echelons. The "JBI platform" for brokering information from a distributed "JBI repository" provides the following core services to "JBI clients": publish, subscribe, query, and control.

A previous effort ("JBI Client Adapter," also known as "Sensor to Shooter Demonstration") included development and demonstration of a set of JBI clients and client adapter components (DCGS Adapter, PDA Adapter, C2PC Adapter, CONUS Adapter, Link-16 Adapter, ADOCS Adapter). This effort integrated disparate existing Command and Control (C2) applications into a JBI information space to demonstrate an operationally compelling scenario. This effort leveraged the Air Force Research Laboratory (AFRL) sponsored and General Dynamics developed JBI Mercury Project which produced a JBI compliant infrastructure product.

The goal of the Mercury Project was to take the initial steps in developing an information interoperability infrastructure to connect disparate software applications or "clients" into a shared information space; and facilitate information sharing between clients for the creation of a system of systems to enable decision makers to obtain the "right" information, at the "right" time, and in the "right" format.

Gemini implements a JBI platform which provides the majority of the same services as Mercury. However, on Gemini we focused on the challenges presented by highly dynamic networks and implementing a relatively lightweight and yet robust architecture while still operating in Mercury's target environment (business class networks).

As an aside, the reader may notice that in other Gemini documentation, including the actual software itself, Gemini is referred to as OIM Gemini. The JBI programs went through a temporary name change to Operational Information Management (OIM). This name was effective for the majority of the Gemini program contract, and so most work products produced during Gemini bear the OIM prefix.

2 Initial Goals

As discussed in the Background section, above, Gemini provides a JBI platform, like Mercury, but which targets decentralized, highly dynamic tactical networks. What goals did we extract from this basic description?

2.1 "Mercury-Like"

Gemini needed to build upon Mercury's concepts and feature set. A client developed to run on Mercury should also run on Gemini's platform with only a minimal set of changes. Specifically, Gemini needed to support the client interfaces described by the AFRL developed Java Common Client API (known as CAPI, available from http://www.infospherics.org/).

Supporting CAPI implied that Gemini must support publication, subscription, and query of Information Objects (IOs or InfoObjects), as well as maintain a registry of InfoObject type descriptors. An InfoObject is simply a container of XML and binary data provided by a client that is bound to a specific type descriptor (type and version pairing). InfoObjects are shuffled along through client-requested sequences, which are bound by the same type descriptor as the InfoObjects they carry.

2.2 Decentralized

Gemini needed to support deployment to a highly dynamic network (or group of networks) where physical nodes come and go and the reliability varies. On such a network, dependence on a predetermined set of "servers" did not make sense. Instead, Gemini needed to bring the services of a traditional message management server into each node. These services needed to be small so that a non-server class machine could use them. And of course, the services on each node needed to be capable of sharing information with each other. Finally, the information sharing needed to be capable of being determined seamlessly during run time.

2.3 Tactical Networking

Gemini needed to operate over tactical networks and thus needed to withstand poor quality network connections which include some combination of low bandwidth, high latency, intermittent or long term drop outs, and out of order packet delivery. Gemini also needed to operate in a mobile, ad-hoc environment in order to support aircraft and mobile personnel communication.

2.4 Scalable

Some tactical networks, such as unmanned aerial sensor networks, are envisioned to potentially contain a large number of nodes (e.g. 10,000) and thus Gemini's design needed to support such a deployment.

2.5 Quality of Service

Because Gemini needed to operate over a decentralized, tactical network, clients needed to be able to express desired service qualities which impact the life span, reliability of delivery, and other controls on the InfoObjects sent or received through individual sequences.

2.6 Simplicity

Because Gemini needed to support deployment to a large number of nodes, a decentralized deployment pattern, and a very dynamic network, Gemini also needed to be simple to install, configure, and maintain.

2.7 Language and Platform Neutrality

Because Gemini needed to be able to be deployed to such a large number of nodes, Gemini also needed to be able to support clients written in various languages running on various hardware and operating system platforms.

3 Trade Summaries

We performed several trade studies during the earlier portion of our project's time frame. Each study resulted in a short white paper which details the findings (available upon request). The research and prototyping we performed targeted narrowing down the goals (above) and the technologies that could have had some potential to help meet those goals. What follows is a brief overview and summary of the findings from each white paper.

3.1 Gemini Operating Environments: UAS and Soldier Systems

Gemini is designed to work across a broad range of systems and deployments. In particular, though, Gemini targets a need for publish, subscribe, and persistence on highly dynamic networks, such as wireless ad-hoc networks. This paper looked into the operating environments expected to be available to Unmanned Aircraft Systems (UAS) and ground troops, in order to determine in what ways we needed to constrain our Gemini technology design and implementation.

3.1.1 Unmanned Aircraft Systems

We found that most of the current Unmanned Aircraft (UA) communicate from air to ground station only, and typically don't fully support Internet Protocol (IP) version 4 communications. According to the UAS roadmap, the UAS of the near future are expected to incorporate onboard gigabit Ethernet IPv6 networking (between modules in the aircraft) and terabyte class storage. They will include full routing outward, including access to multiple other aircraft and the Global Information Grid (GIG). On these UAS, Gemini may be installed in the form of a small payload or onto an existing computing platform within the UA.

The class of UAS known as Micro Air Vehicles (MAVs), which are extremely small, are highly unlikely to carry such a rich set of technologies. Since these have exceptionally small payload capabilities (often in the range of 0.1 pounds), and limited power, it is unlikely that Gemini can be installed onto such a platform. We would instead deploy Gemini within the ground station portion of the MAV.

3.1.2 Soldier Systems

Future Force Warrior (FFW) is a ground soldier system which will eventually replace the current Land Warrior systems. FFW soldiers carry a small computer with modern, low-end consumer capabilities. These systems have run Windows XP SP2 and Red Hat Enterprise Linux. They utilize a network device which provides 1 megabit per second throughput over a UDP multicast-only mesh network. This finding is largely compatible with what we discovered for UAS, but with an added clue that our solution ought to work in environments where networking is restricted to multicast and where network throughput is low to moderate.

The warrior network also utilizes a wide variety of Unattended Ground Sensors (UGS). The various UGS are also using a variety of connectivity options: some are specialized interfaces, some achieve their data transfers over a ZigBee 802.11b mesh network, and other larger sensors connect via wired Ethernet or Iridium satellite links. The various UGS employed by FFW would require more investigation to determine feasibility of enabling Gemini connectivity directly to them, but after a cursory check, Gemini deployment to these devices has not yet been ruled out.

3.2 Network Dissemination and Discovery

As we've discussed, Gemini provides publication and subscription services across decentralized, ad-hoc networks. The total peering capability of these networks may be very large. And, each network or access point may carry restrictions; for example, to specific IP protocols or ports. Additionally, as Gemini peers come online, they need to be discovered by other peers.

In this study, we examined how several different off-the-shelf networking products and tools may help us resolve our network dissemination and discovery needs. We focused on these products and tools: PrismTech OpenSplice 2.0 DDS, RTI Data Distribution Service 4.1d, OCI DDS for TAO (OpenDDS), Multicast Dissemination Protocol, Network Programming with ACE, and the Java Message Service. A chart which details our findings is included in Appendix A: Network Dissemination and Discovery Product Feature Comparison.

Products were eliminated during this trade due to their immaturity, expense, lack of decentralization, lack of scalability, and lack of flexibility (e.g. supporting streaming reliable delivery from one node to many, but no unreliable mode or acceptance of nodes disappearing for extended periods and then reappearing).

The study narrowed our selection to RTI DDS and a custom in-house solution. In the end, we eliminated RTI DDS for several technical reasons that we would need to work around using custom code; for instance: its inability to relay data across network connections, complications in getting it to survive network interface drop outs, and the extensive tweaking it requires to operate efficiently for each type of network to which it is connected. In addition, RTI DDS was quoted at a cost of approximately $23,550 annually for development plus $800 per runtime license (run time costs go down significantly when in very high quantities).

3.3 XML Compression

A portion of the Information Objects (IOs or InfoObjects) that Gemini clients publish may include Extensible Markup Language (XML) metadata. Because Gemini nodes may be operating over bandwidth restricted network connections, we performed a quick study of the available XML compression technologies to determine if one of these products would provide us with an elegant and efficient way to compress a client's XML metadata prior to transmitting this data.

There are several technologies specifically designed for compressing XML data (for example XGrind, XML-Xpress, and XMill). A general purpose compressor, gzip, is built into Java, and when compared to the other products, it was extremely simple for us to support gzip compression in our code. By using gzip, Gemini may also compress binary payloads. So, Gemini uses gzip.

3.4 XML Database

Clients publishing IOs may request persistence of the publications. This permits other clients to query for information that was published in the past. Gemini clients filter query results by type and version. They may further filter results to those which match an XPath predicate. (XPath is a language for addressing or selecting portions of an XML document, among other things.) We performed a quick study to determine which XML database products are currently available that would meet Gemini's needs.

Many XML databases exist. We eliminated those products with poor or lacking documentation, poor support, lack of XPath support, or lack of ability to run in an embedded (e.g. library) mode. This narrowed our comparison down to Xindice, eXist, and Berkeley DB XML. Berkeley DB XML carries either a cost per run-time, or a negotiated one time cost for distribution under a single contract (expected to be $40,000 or more). We eliminated Berkeley DB XML due to the high cost. In the end, we selected eXist because it is a freely distributable XML database with which we already have in house familiarity due to our use of eXist on Mercury.

As an aside, while performing the research and prototyping for the network dissemination trade, we found that the commercial DDS products from PrismTech and RTI have counterpart products which handle persistence. However, their products were not appropriate fits for Gemini. Besides the prohibitive cost, they do not offer XPath support which is needed to implement the JBI CAPI. They also don't provide the style of partially redundant data distribution that we wanted for Gemini (i.e. they don't work well in an ad-hoc networking environment).

3.5 SQL Database

We decided that a SQL database would best serve as the basis for the management of type descriptors, much like Mercury. Gemini stores all information about type descriptors into the SQL database; the actual schema definitions are stored as files on disk, referenced within the SQL tables.

Although we did not conduct a trade study on SQL databases, we used the same process in deciding the SQL database as we did in deciding the XML database: we wanted a zero-cost, easy to configure, fast, and, most important, embeddable database implementation. We felt that it is important for Gemini Node Service to use an *embedded* database since the node service is intended to be installed everywhere and our experience on Mercury tells us that external database products burden users with separate installation and maintenance steps. We chose SQLite as our database implementation because it is very small, is extremely fast (it is written entirely in C), we had prior experience using it on other projects, it is available on just about every platform, there are available Java API bindings, it is licensed in the public domain (zero-cost), and it is strictly an embedded database.

4 Narrowing Our Focus

As we worked through the trades, system requirements, and systems design, we continued to narrow the focus for Gemini even further. We realized that, at least for this initial contract, we must target a more specific set of devices and include a more restrictive set of features. In this way we attempted to "keep things simple" where possible in order to focus on quality over quantity and to stay within our tight budget and schedule constraints.

Here are some of our critical focus decisions.

4.1 Implement Node Service in Java Programming Language

The developers initially on the Gemini team had more overall familiarity with Java then with other languages. We did do some early prototyping in C# (.NET) and in C++. But, in order to make the most efficient use of our programming skill sets, we decided to stick with Java.

Java also makes it relatively easy to build Gemini and run it on multiple platforms. Alternatively, we could have used C++ for this, with the Adaptive Communication Environment (ACE) libraries and other cross-platform programming libraries. The .NET runtime environment, on the other hand, is only well supported on Microsoft Windows platforms, with relatively weak (or inconsistent) support of its standard runtime libraries on other platforms such as Linux.

We briefly checked out the Java Mobile Edition technologies. We quickly eliminated them as an option because their APIs aren't even a proper subset of the Standard Edition APIs (for instance, classes related to networking are vastly different). We did not want to implement multiple, separate code bases for the server.

We chose Java 6.0 (Sun's version 1.6.0u2) for our implementation of the Gemini Node Service. This choice was based largely on improvements to the Java collection libraries, Swing APIs, and performance enhancements to the runtime. As you will read later (in section 5.4, "Client Language and Platform Independence"), we also knew that Java 6 would not preclude us in the future from providing client-side libraries for older Java versions or other programming languages.

Java 6 doesn't run on embedded platforms, such as those based on the ARM or PowerPC processors. From research, we know that newer "embedded" platforms, such as the small, Intel based PCs that Future Force Warriors will carry, are running full, modern operating systems which are capable of running Java 6. Some modern platforms, such as Mac OS X, do not yet support Java 6, but we expect that they will in the near future. In addition, Sun is in the process of open-sourcing Java and once well-established, will likely be ported to a multitude of platforms.

4.2 Multicast Makes Sense

Given that Gemini's primary target environment is a large scale network based on radio transmissions, and that the publish and subscribe mechanisms which are at the heart of any information space are based on many to many relationships, then it seems logical that we would make best use of radio bandwidth by sending a message once and having all the other nodes "listen in" and receive it. Radios by their very nature work this way anyway, since the air waves are shared between all in range users (and not switched, like the typical modern, wired network).

The multicast mode of User Datagram Protocol (UDP, which is a layer of IP) is designed for the many-to-many communications that we've just described. This ought to be a good fit for large scale mobile networking, to reduce overall transmission redundancy. So by default, Gemini Node Service discovers and communicates with other Gemini nodes via multicast.

4.3 MANET is not Our Challenge

Many researchers are looking at the problem of standard network services for devices which are initially aware only of themselves, and which may frequently gain or lose access to other such devices. Such a network is known as a Mobile Ad-Hoc Network (MANET). The self forming, fundamental network services (e.g. IP routing, multicasting) which these researchers are currently exploring are intending to form what is referred to as a MANET overlay, a type of mesh network.

We found that a great deal of research is underway, but none of the software based MANET overlays that we found (as of late 2006) was mature or capable of handling more than roughly forty or so nodes in the MANET. Cisco is working on mesh network hardware which once available, may work well, and some other groups, like ITT (http://acd.itt.com/products.html), have very small radios already deployed, but they don't yet scale to a large number of nodes. We invited Professor Violet Syrotiuk of Arizona State University to present her group's MANET research (see http://mars-lab.eas.asu.edu/). This only served to confirm our understanding that the field is still "new" and largely unsolved, and that the problems are quite difficult.

So in the end, Gemini does not even attempt to solve the network stack level issues of MANET overlays. It *should* operate well over a MANET overlay which is supportive of typical networking needs. Gemini is built with this type of network in mind, so it is supportive of the idea that connections to other nodes are transient, lossy, and frequently changing. For the version of Gemini delivered on this contract, this means that the MANET must support UDP and ideally would support the routing of IP multicasted data (see "Multicast Makes Sense," above; otherwise, we also support UDP unicasting).

4.4 Network is Our Most Scarce Resource

Gemini focuses on the network. We already know that the JBI information space ("Infosphere") concepts work for enterprise class deployments. Sensor-to-Shooter I and II demonstrated this using JBI Mercury as their platform. With Gemini, our new challenge is the users with mobile, radio based network connections, such as aircraft, ground vehicle, sensor, and soldier-integrated deployments.

While we expect that network bandwidth will continue to improve, we also know that the number of nodes on the network will greatly increase. Since the node service must run on every mobile node (see section 2.2, "Decentralized"), the risk is that we over saturate the network with the control messages that we require to discover nodes, recover lost packets, etc. So on Gemini, the greatest challenges we faced involved our management of network resources to share the InfoObjects produced by clients in a reliable and efficient fashion.

5 Technical Accomplishments

5.1 Architecture

Gemini consists of several executable packages: node service, Mercury bridge, administration tool, and test tool. Gemini also contains a client library, which client developers may use to connect to the node service from Java. Any combination of these packages may be installed onto a given host machine.

Gemini Node Service, which we also refer to as "Gemini Server," is a single process comprised of several component areas, as shown in the following diagram (Figure 1). As our team developed Gemini, this componentization within the node service process ensured that we kept its functional areas separate with clean divisions between the roles of each component area. Exposing each component with Java interfaces made concurrent development easier and should also ease future maintenance.

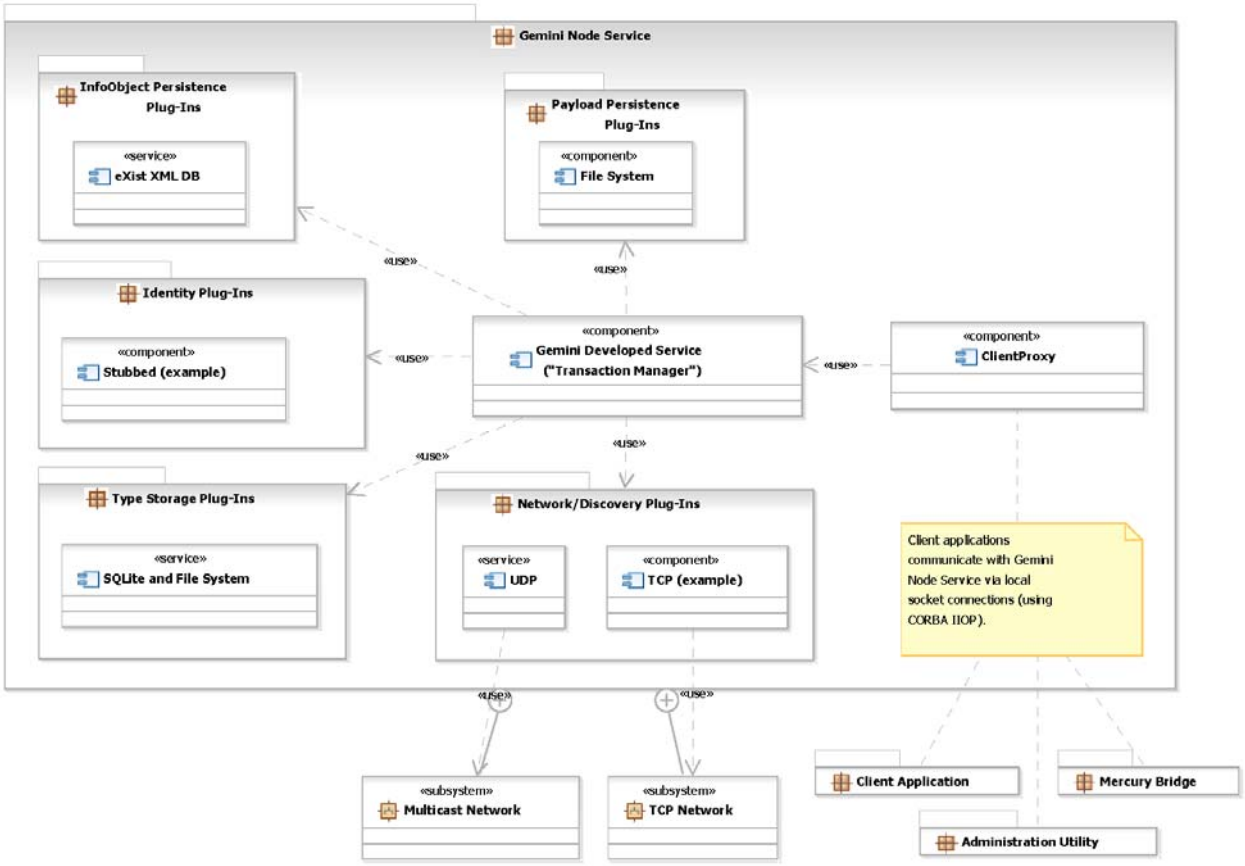


Figure 1: Gemini Node Service Architecture

During startup, the node service reads configuration data (Java property files) which define its behavior and capability set. It also locates plug-in Java archives (JARs) which connect the node service's logic and management components to external data and processing sources. The following section goes into further detail on Gemini's extensibility.

5.1.1 Packages

The Gemini core software contains several executable, library, and interface packages. Table 1 documents the various packages included in the Gemini core software distribution.

Table 1: Gemini Core Software Packages

Package Name	Description
Node Service	The Gemini Node Service is the process, potentially executed at several locations on the network, that hosts client connections and provides them with publish/subscribe/query functionality. It is comprised of several Java JAR files (gemini_server.jar, gemini_ior_exist.plugin.jar, gemini_ior_filesystem.plugin.jar, gemini_msr_sqlite.plugin.jar, gemini_net_udp.plugin.jar, gemini_sec_stub.plugin.jar), some COTS libraries, and configuration ("property") files.
Java Client Library	The Java Client Library, gemini_client.jar, abstracts the CORBA interface to the Gemini Node Service and is the preferred method for Java clients to use to connect to the Gemini Node Service. The client library implements the Common Application Programming Interface (CAPI) version 1.5 as documented at the Infospherics web site http://www.infospherics.org/api/CAPIv1_5/docs/index.html.
Test Client	The Test Client, gemini_test_client.jar, provides a graphical user interface that can be used to exercise much of the Gemini Node Service functionality.
Administration Utility	The Administration Utility, gemini_admin.jar, provides a graphical user interface for administrating the Gemini Node Service. The utility functionality is currently incomplete.
Mercury Bridge	The Mercury Bridge, gemini_mercury_bridge.jar, allows Gemini to be bridged with Mercury, another JBI/OIM implementation developed by General Dynamics C4 Systems. The bridge publishes all Gemini published InfoObjects into Mercury and vice versa. The bridge also allows querying Mercury whenever a Gemini distributed query is performed.
Java Sample Client	The Java Sample Client is buildable source code of a simple Gemini client that developers new to Gemini can use as reference in developing Gemini-based applications.
CORBA IDL	The CORBA IDL definitions that can be used by a non-Java developer to interface with the Gemini Node Service.

5.1.1.1 Graphical Test Client

The test client utilizes the Gemini client library, as we expect that most Java based clients will. It exercises the core of the CAPI and Gemini specific interfaces, (which is everything except for administration, query provider registration, and type management).

The test client permits connection and authentication to the server. From that point, the user may work with publisher, subscriber, and query sequences, which each have their own categories on the main user interface. The main panel also shows some top level statistics, as shown in Figure 2.

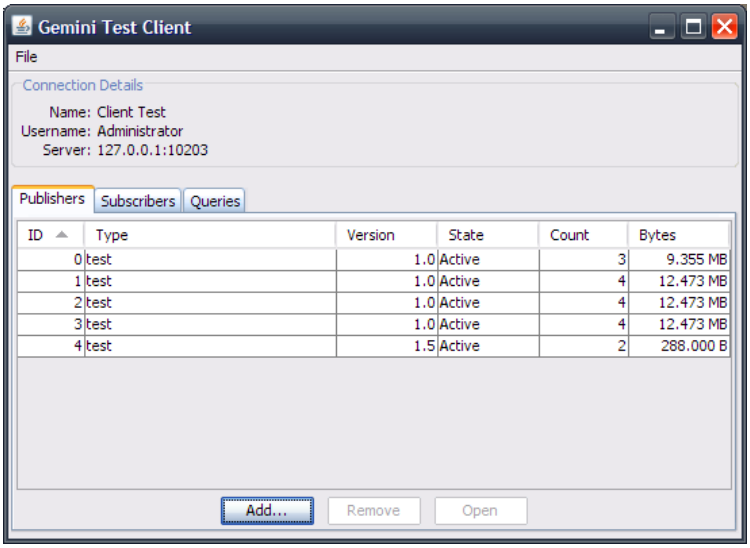


Figure 2: Gemini Test Client – Main Panel

When adding a new sequence, a dialog is presented with extensive options which are appropriate to the sequence being created. For instance, while creating a publisher sequence, you can create a single publisher sequence which publishes each time you press a Publish button, or you can create a bulk set of publishers which each publish the number of times requested with a requested delay between publications. And for each publisher, you may specify the location of metadata and payload, and QoS options (persistent, reliable delivery, compression, XML well-formed, XML validated against schema). This is shown in Figure 3.

Once sequences are created, they remain until removed or until the test client is closed. All of this combined flexibility in the test client, while being easy to use, allowed us to simulate various behaviors that real clients may attempt as we were testing the logic throughout the node service.

The test client is part of the Gemini installation package because it will be useful for end user developers and administrators. Developers may use the test client to stand in for other clients which are in concurrent development. This way, the developer may simulate some other client by creating the appropriate publisher to send data or the appropriate subscriber or query execution to verify receipt or storage of data. Administrators may use the test client to validate an installation or multiple system deployments of Gemini.

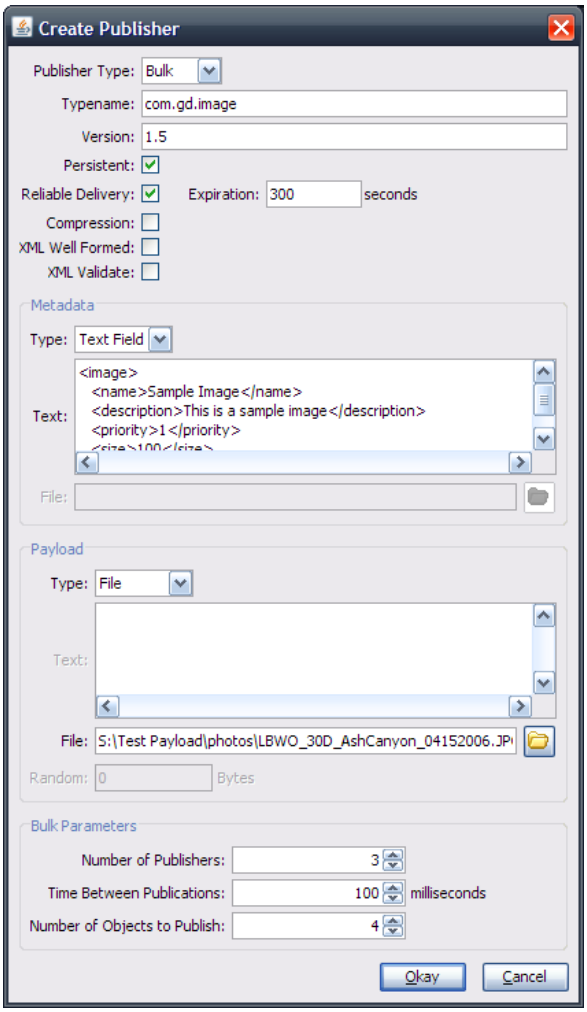


Figure 3: Gemini Test Client – Create Bulk Publisher

5.1.1.2 Administration Utility

The administration utility is incomplete but supports the following features: register a type descriptor, remove a type descriptor, delete an InfoObject, shutdown the node service, view all connected clients, and view all publishers on the local node. Some of the missing functionality includes viewing all local and remote publisher, subscriber, and query sequences; being able to snoop in on message activity to and from a node and/or a single client; view internal node service statistics of both local and remote nodes; graphically view and manipulate the entire type repository; control and configure authentication; etc. A screen capture of the administration utility is provided in Figure 4.

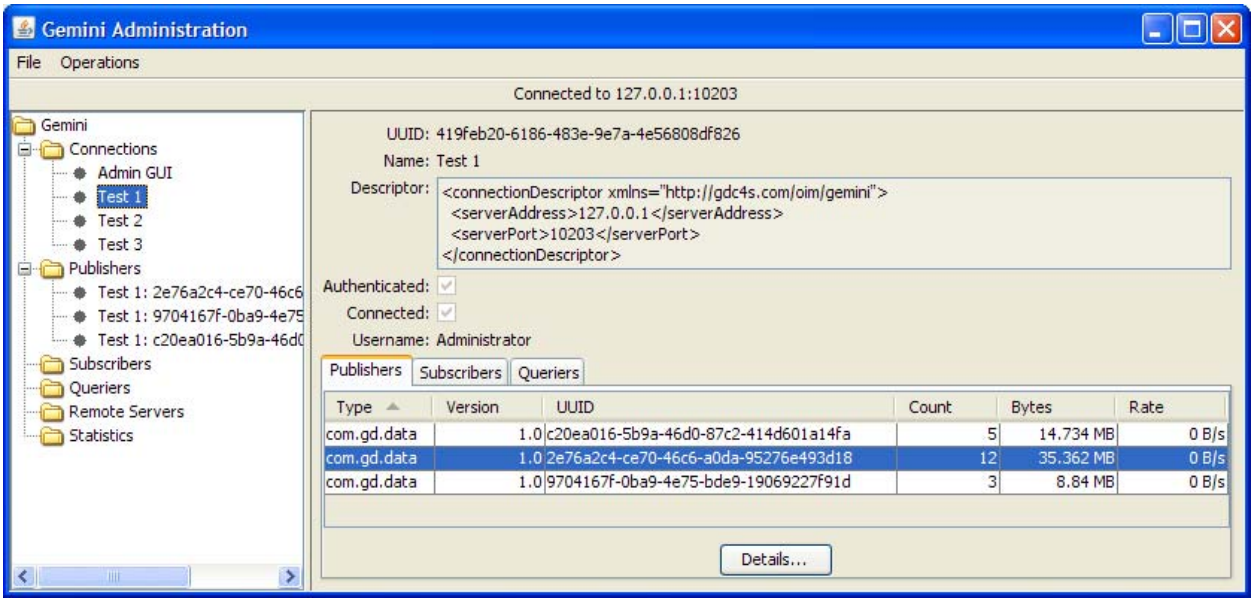


Figure 4: Gemini Administration Utility

5.1.1.3 Java Sample Client

The Gemini installation package includes sample client code. The sample client uses the Java client library to connect to Gemini. It is heavily commented and intended to be used by developers who are just learning to use Gemini (and even those just learning to use the CAPI). The sample code covers basic tasks involving subscriptions, queries (local and distributed), and publications. It is run from the command line, and takes command line arguments to alter its behavior (thus demonstrating variations on the basic themes). Comments in the code explain alternative ways of accomplishing certain tasks.

We wanted to also provide a sample client written in C# (.NET) using IIOP.net to communicate to Gemini Node Service. However, we did not have sufficient time to start this task.

5.2 Extensibility

Gemini Node Service ships with embedded copies of eXist (an XML database) and SQLite (a relational database). Actually, though, both of these products are bundled into "plug-ins," or replaceable libraries which the node service detects and incorporates when it starts up. Using plug-ins forced us to ensure separation between the core server components and input/output services that may need to be replaced for a deployment's specific needs. The following table lists the extension points along with the default plug-in for each.

Table 2: Node Service Extension Points

Extension Point	Description	Bundled Plug-In
Type Registry Backing Store	storage/retrieval of optionally registered types and associated XML schemas	combination of SQLite (type registry) and file system (XML schema document storage)
InfoObject Backing Store	storage/retrieval of persisted information objects; actually two plug-ins, one for XML data and one for binary data	combination of eXist (XML storage) and file system (binary storage)
Networks (concurrent, varied instances are supported)	server to server communications; manages collection of logical connections (e.g. for a TCP server or UDP unicast)	UDP; supports unicast and multicast modes
Security	provides authentication of connected clients and remote node services; authorizes client actions	stubbed; all credentials are approved, all actions are authorized; user "Administrator" has administrative access

5.3 Ease of Installation and Execution

The complaints we hear most often from users of JBI Mercury center around how difficult it is to install and get running. Mercury requires up to six additional software products to be installed, and involves extensive post installation procedures.

With Gemini, we've gone out of our way to make the installation experience as simple and straightforward as possible. All of Gemini's components and dependencies are bundled into a single installation package. The only required pre-requisite is Java 6, and on the Windows version of the installer, the user will even be given assistance in downloading Java 6 if it isn't already installed. (Installation of Mercury is also a pre-requisite if the user chooses to use the Mercury Bridge application.) As shown at right, the installer allows users to select which components of Gemini will be installed, and to configure some of the options of those components. Additional configuration options are presented if the user selects the "Advanced Configuration" option.

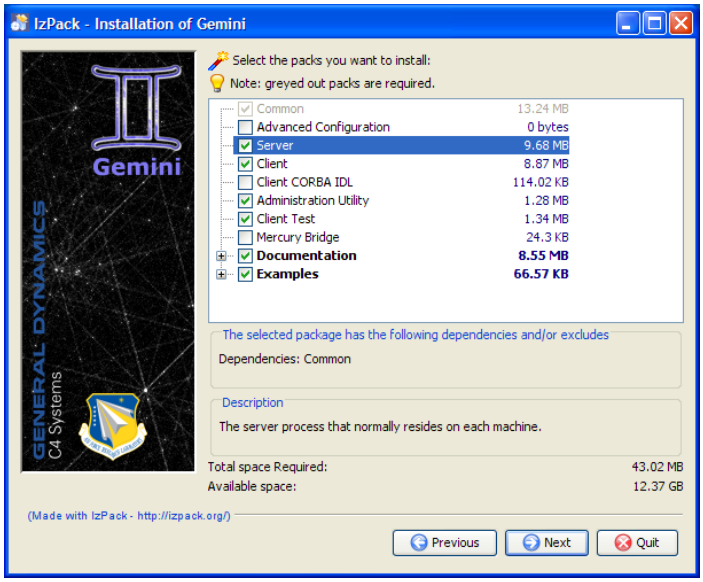


Figure 5: Gemini Installer

Once installed, Gemini Node Service runs as a Windows or Linux boot-time service (by default). Shortcuts are placed on the Windows Start menu or the Linux GNOME menu to help users more easily stop and start the service, view its log file, run the administration tool, etc.

5.4 Client Language and Platform Independence

Mercury is directly accessible via Java only. This is because Mercury clients connect to Mercury via a combination of Java specific technologies. With Gemini, we sought to open up this access, enabling connectivity to the server without requiring the use of Java.

5.4.1 CORBA via IIOP

When we considered platform and language independent inter-process communication (IPC) mechanisms, only two really came to mind: Common Object Request Broker Architecture (CORBA) and web services. Web services are generally slow, encumbered by XML overhead, and generally need to run inside of a web container. They are also difficult to deal with from C++, which is still a very popular language. CORBA is a mature infrastructure that has many implementations for various languages and platforms, many of which are zero-cost and open source. So, we selected CORBA for the Gemini client to node service IPC. Specifically, our Object Request Broker (ORB) traffic is carried over Internet Inter-Orb Protocol (IIOP), which is the standard implementation for CORBA over TCP/IP.

In our previous experience, CORBA performs quite well under load and provides high throughput. In tests with Gemini between two machines connected via gigabit Ethernet, we currently observe a throughput of over 60 megabytes per second between a publishing client and a separate subscribing client. Gemini uses JacORB, an open source and zero-cost Java ORB (Object Request Broker), to help us accomplish this speed. Earlier in development, we were utilizing Sun's ORB which is built into Sun's Java 6 Standard Edition. However, maximum throughput was much lower (on the order of 8 megabytes per second) while utilizing a lot of the processor, so that is why we switched to JacORB.

5.4.2 Java Client Library

We developed a Java client library which is included with Gemini. The client library implements the CAPI. Calls into these interfaces are proxied to the server, via CORBA, and the response is proxied back. So, the CORBA interface definitions used by Gemini Node Service bare close resemblance to the CAPI (discussed later). The differences are mainly limited to Java or CORBA Interface Definition Language (IDL) concepts which are incompatible.

Our Gemini interfaces also include additional methods and interfaces, which we refer to as CAPI extensions. These methods permit client developers to have access to features which are not described by CAPI, such as quality of service parameters, distributed query execution, and query provider registration. We also define administrative interfaces, which are accessible to properly authorized clients and used by the administration tool.

5.4.2.1 CAPI Compliance

Gemini is CAPI compliant. That is to say, Gemini's Java client library implements all of the CAPI interfaces, and provides a default (zero argument) constructor on a concrete implementation of `ConnectionManager` so that generically designed clients may use Gemini without code changes.

A client may restrict itself to utilizing only the base CAPI interfaces, and still access many of the Gemini specific features. Here is a code snippet example which uses the CAPI generic concept of sequence attributes to specify to the Gemini Node Service that InfoObjects received on a particular subscriber sequence must contain well-formed (syntactically valid) metadata XML:

```
org.infospherics.jbi.client.SubscriberSequence subSeq = null;

// ... create subscriber sequence ...

// Validate that received InfoObject metadata contains well formed XML
String attributeXml =
   "<validationAttribute xmlns=\"http://gdc4s.com/oim/gemini\">" +
   "   <validateXmlIsWellFormed>true</validateXmlIsWellFormed>" +
   "   <validateXmlAgainstSchema>false</validateXmlAgainstSchema>" +
   "</validationAttribute>";
subSeq.setSequenceAttribute("Validation", "1.0", attributeXml);
```

The Gemini client library also includes XMLbeans generated bindings to the XML schemas which define the recognized attributes, connection, authentication parameters, and the platform-specific portion of the InfoObject ("extended metadata XML"). These bindings permit client developers to use Java objects to specify the XML parameters, instead of needing to write the XML by hand (as was done above).

5.4.2.2 Enhanced Access via Casting

We suspect that many developers would find that configuring their client connections and sequences via XML is a chore, even with the auto generated Java/XML bindings. So, for every possible generic attribute XML, we implemented corresponding concrete methods in both the CORBA IDL and the Java client library. In the following code example, we achieve the same result as above except that we use casting to access the Gemini specific method which corresponds to the validation attribute:

```
org.infospherics.jbi.client.SubscriberSequence subSeq = null;

// ... create subscriber sequence ...

// downcast to more specific Gemini sequence
com.gd.c4s.gemini.client.impl.SubscriberSequence geminiSubSeq =
   (com.gd.c4s.gemini.client.impl.SubscriberSequence)subSeq;

// Validate that received InfoObject metadata contains well formed XML
geminiSubSeq.setValidationAttribute(true, false);
```

Clients must use Gemini specific client interfaces to access Gemini's administration, distributed query, query provider registration, and XPath namespace prefix mapping capabilities. With the exception of XPath namespace support, these Gemini services are far beyond what would be capable through XML attribute specifications. XPath namespace support, though, could be accomplished via a generic sequence attribute – and we've documented this as an enhancement idea. We simply didn't have time to implement it.

5.4.3 Other Languages and Platforms

Gemini is built and tested on Windows and Linux with Java. Currently, the node service must be deployed to Windows or Linux. Clients may connect to the node service from any machine that has TCP connectivity to it. Clients may be written in a variety of languages, using an appropriate CORBA implementation for the language. IIOP.net (http://iiop-net.sourceforge.net/) is a great zero-cost and open source option for .NET (e.g. C# language) developers. TAO (The ACE ORB, http://www.cs.wustl.edu/~schmidt/TAO.html) is a popular, zero-cost and open source option for C++ developers. ORBit2 (http://www.gnome.org/projects/ORBit2/) is a zero-cost, open source option available to C developers and has bindings for C++, Python, Perl, and others.

The CORBA Interface Definition Language (IDL) which describes Gemini's client and server interfaces and methods is packaged with the Gemini installer. We fashioned the IDL to look a lot like the Java CAPI, except modified as needed to be appropriate for CORBA. The IDL is in two sections, a CAPI section which is the basic interfaces from CAPI, and a Gemini section which defines Gemini's extensions to the CAPI. The Gemini section includes other Gemini specific features, such as the node service administration interfaces.

Here is a sample of the IDL, from `capi/PublisherSequence.idl`, which shows how clients publish InfoObjects using a publisher sequence. It is extremely similar to the CAPI `PublisherSequence.publishInfoObject()` method:

```
#include "Exceptions.idl"
#include "InfoObjectSequence.idl"

module capi
{
   interface PublisherSequence : InfoObjectSequence
   {
      string publishInfoObject(in InfoObject io)
      raises(exceptions::PausedSequenceException,
         exceptions::PlatformFailureException,
         exceptions::PermissionDeniedException,
         exceptions::SequenceStateException);

      // ... additional methods not shown here
   };
};
```

In this next sample, we show how the publisher sequence is extended to add in Gemini specific features. In this case, the snippet shows how publication reliability may be specified by a client:

```
#include "PublisherSequence.idl"
#include "GeminiInfoObjectSequence.idl"

module gemini
{
   interface GeminiPublisherSequence : capi::PublisherSequence,
      GeminiInfoObjectSequence
   {
      void setReliableDeliveryAttribute(in boolean reliable,
         in long long expiration)
      raises(capi::exceptions::PermissionDeniedException,
         capi::exceptions::PlatformFailureException);

      // ... additional methods not shown here
   };
};
```

Note that the `GeminiPublisherSequence` interface inherits from the CAPI `PublisherSequence` interface. Like in Java, direct CORBA clients *narrow* (similar to an object cast in Java) the `PublisherSequence` interface reference to a `GeminiPublisherSequence` reference to access Gemini's extensions.

The figure below demonstrates a possible deployment scenario using different CORBA implementations with clients both internal and external to the machine running the Gemini Node Service.

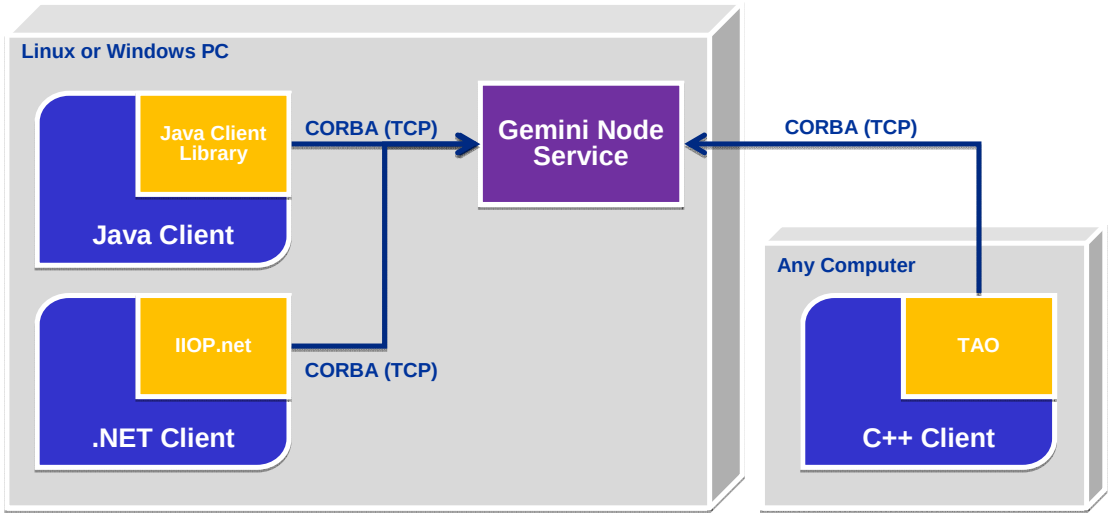


Figure 6: Example CORBA Deployment

5.5 Quality of Service

We've touched on the ability for clients to request that received InfoObjects contain well-formed metadata XML. This is one of Gemini's configurable service qualities. Gemini's quality of service (QoS) options fall into two categories: static boot-time (node service) configuration and dynamic per-client sequence configuration. We'll now detail each of these.

5.5.1 Node Service Configuration

The node service reads configuration data files during startup. These modify the behavior of the server for the duration of its run time. For instance, selection of network plug-ins, as well as their respective configurations, is determined during startup with help from the configuration files. We thought about various deployment environments and exposed the relevant configuration options for those within the "Advanced Configuration" panels of the Gemini installer. For example, the installer defaults to using multicast over all multicast-capable network interfaces, which may not be acceptable.

There may be times where an administrator needs to reconfigure the node service after installation, or times in which options need to be modified which are not accessible via the installation panels. For instance, the installer allows up to two logical networks to be configured. If more are required, then the configuration files must be modified manually using a text editor.

Here are some of the more interesting, service quality related, node service configuration items:

Table 3: QoS Related Node Service Configuration Items

Category	Item
Client	default list of namespace-prefix mappings
	maximum permitted metadata and payload sizes
	enable metadata is well-formed XML validation (client may override)
	enable metadata validation using registered schema (client may override)
	default publication reliability mode and duration

Category	Item
Discovery	node and sequence announcement frequencies
	node identifier retention
	positive ack. ranges per node announcement
	inactive node and sequence aging timeout periods
Messaging	maximum message block size
	maximum node service hops (for message and retransmission relays)
	maximum number of processed message hash cache entries
	maximum number of retransmitted block hash cache entries
	unreliable message out-of-order receipt timeout
	delay before sending retransmission requests on partial messages
	maximum delay before response to retransmission request is sent
	durable message, expiration cache size
	hashing algorithm
	enable message compression
	enable negative acknowledgement responses
	enable positive acknowledgement responses
Network (and UDP plug-in)	list of instantiated networks
	mode (unicast, multicast)
	transmit queue size
	socket send/receive buffer sizes
	multicast hops (time to live)
	enable reduced relay mode
Persistence	enable local persistence store
	enable distributed persistence
	enable distributed query response
	number of volunteer nodes per persisted sequence
	distribution coordination message frequencies/delays
	maximum disk usage for metadata and payload storage

5.5.2 Client Specified, Sequence Configuration

Client applications may set additional service qualities on a per-sequence basis. An example of this is shown in "5.4.2.1 CAPI Compliance," above. Most service qualities may be set through standard CAPI sequence attributes by providing the desired sequence attribute information in a prescribed XML format. A second example, "5.4.2.2 Enhanced Access via Casting," is also shown. It achieves the same resulting service quality as the first example, but uses Gemini interface extensions instead of the sequence attribute XML approach. The following table describes the client specified service qualities understood by Gemini Node Service. All QoS options are accessible via Gemini interfaces, and those with check marks are also accessible via XML sequence attributes.

Table 4: Client Specified QoS Options

Category	Item	May Use XML Sequence Attributes
Publish, Subscribe, and Query	ensure InfoObject metadata is well-formed XML	✓
	ensure InfoObject metadata is valid instance according to registered XML schema	✓
Publish	persist publications	✓
	enable reliability (when enabled, a reliability expiration is also specified)	✓
	compress publications	✓
Query	distribute query execution to entire Infosphere; asynchronously provide results	
	register query provider, which is given opportunity to respond to distributed query requests	
Connection	notification when node service is shutting down or goes offline	
Administration	register for notification when connections and sequences created/updated/removed	
	receive periodic statistical updates	
	snoop in on InfoObjects traversing a specified sequence	
	retrieve list of connections or sequences on demand	
	shut down the node service	

5.6 Network Transmission Management

Gemini nodes help other Gemini nodes to receive data in three different ways. First, nodes lose data due to computer, network, and transmission limitations, and other nodes help the disadvantaged nodes to recover the lost information. Second, some of the nodes are configured with multiple network interfaces, and can relay data between these interfaces. Finally, nodes share a common distribution of certain information, such as persisted InfoObjects.

These features presented a challenge to the system's scalability. In order to ensure that a Gemini Infosphere may scale to very large sizes, we needed to greatly limit the duplicative incidents of packet transmission, node service processing, and client message delivery.

5.6.1 Lost Information Recovery

Some of the internally used Gemini node-to-node messages are sent reliably or durably (see "Delivery Modes," below on page 23). Client publications are transmitted reliably, by default. (The default publication mode may be changed to unreliable, in which a single delivery attempt is made; and, client publishers may always override the current delivery mode using a quality of service attribute on the publisher sequence). Gemini is designed to cope with unreliable network connections. Several mechanisms are in place to recover lost information while remaining decentralized in nature, and reducing the risk of having many nodes request or recover the same information at the same time.

First, before a message is distributed to any network, a cryptographic (MD5) hash is calculated on the body of the message. This hash is cached and the message is transmitted. Nodes receiving the message will add the message's hash to their own cache. In addition, they will relay the message back out to any network connections serviced by that node on which the message hasn't yet traveled. We will describe some of the ways in which the message hash is used in the following subsections.

5.6.1.1 Negative Acknowledgement

When messages are published to the network, the contents of the message are first encoded into an array of bytes. Segments of these bytes are then divided into message "blocks" which are each then placed into an "envelope" and sent to the network as datagrams (in the case of the UDP network plug-in). When a message block is received into a node, the message management portion of the node service will locate a corresponding partial message for the block and add it in. If no further blocks arrive for the partial message after some configurable period of inactivity, then the node service will detect that the message is incomplete and will issue a retransmission request for the missing blocks.

This is relatively straight forward: a node receives some parts of a message, detects the parts it is missing, and requests them. But how can this scale? What if many nodes are missing the same blocks? What if many nodes have the missed block and are willing to resend it? It would be wasteful for all of them to make the same request or response over a multicasted network.

To resolve this concern, each node listens to the retransmission requests made by others. A node in need will only request retransmission if another node hasn't recently made the same request. Nodes responding to a request will delay the response by a small random amount: the node only retransmits the missing message blocks if no other nodes have responded during the random timeout period. This cooperative style of recovery requires the nodes to listen to the messages (requests and retransmissions) from other nodes. So, multiple nodes may end up issuing requests or recovery blocks at around the same time, particularly when network latency is high or the processors are busy. The random delays greatly reduce this occurrence, but we have ensured that extra responses are not harmful in terms of the recomposition of a message. If a stray retransmission block arrives after the message was fully recomposed and removed from the temporary recomposition cache, the stray block will be correctly and completely ignored – a processed message hash cache is used to ensure that no message is processed "as new" more than once.

5.6.1.2 Positive Acknowledgement

Negative acknowledgements report what was missed. If the node service misses an entire message, then no negative acknowledgements will be sent – the software wouldn't know that anything was missed. So, we also implemented a positive acknowledgement system on Gemini.

Positive acknowledgements are attached to node announcement messages (these are the periodic messages which allow nodes to discover each other and to detect when nodes have disappeared). These acknowledgements work by reporting the serial numbers of reliable blocks received from other nodes, and then having those nodes determine what we *didn't* (but should have) reported as received. Nodes respond to positive acknowledgement reports just as they do for negative acknowledgements: by collecting together the blocks which are determined to have been missed, scheduling transmission of these blocks after a random delay, and then transmitting

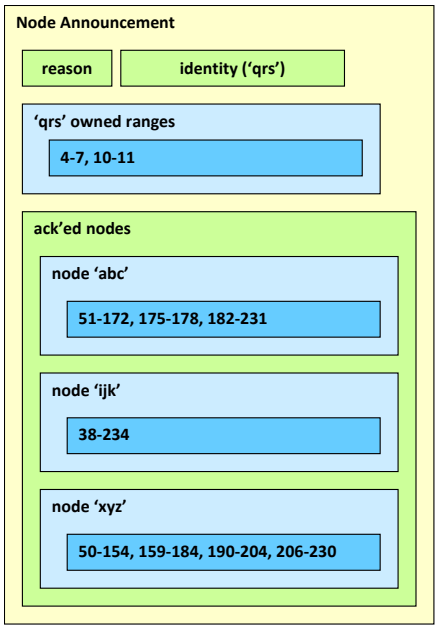


Figure 7: Node Announcement Organization

those blocks which weren't already transmitted by other means for that perceived "request." A simplified, graphical view of an example node announcement message is shown in Figure 7.

Implementation of negative and positive acknowledgements proved to be more complicated and processor intensive than what we originally thought. These are complicated mainly due to the algorithm for avoidance of excessive retransmissions. We end up having to keep track of the last time an individual block has been transmitted, the last time it was requested, and whether or not it is in queue for retransmission. These details are needed because Gemini supports nodes coming and going with high frequency, and wants to reduce network bandwidth usage. Therefore, the block recovery algorithms use information about what we know instead of using some sort of recovery election process or point to point style of recovery. In addition, the task of recovering lost blocks is spread among all of the connected nodes, so that the original sender of the message is able to go out of the network and recovery may still take place. This is the primary reason why recovery responses are delayed by random amounts.

Part of the scalability of a network of Gemini nodes is directly affected by the node announcement frequency and size. Node announcement frequency affects the latency involved in the recovery of entirely missed messages as well as the detection of other nodes, so, although the frequency is configurable, it should be kept fairly low on the order of seconds. Node announcement messages can also become lengthy on a lossy, busy network; however, their size is capped by a configurable amount in order to prevent network and processor overloading. So, on a network with little bandwidth, error recovery has the potential to use a significant portion of the bandwidth. Reduced relays, covered in great detail in section 8.5.1, can greatly help this situation.

5.6.2 Connection Relays

Gemini Node Service may be configured to have multiple network connections, which may even be on different network interfaces. For example, the node may be configured to communicate with two separate wireless networks as well as an Ethernet wired network. Or the connections might be on the same physical interface, such as two multicast connections (using different group addresses) on the same interface.

In either case, when a message (such as an InfoObject publication) has been completely received into a node for the first time (i.e. all message blocks have been received and processed), the node service will then resend the message to each of the network connections which did not participate in the original receipt.

The message relay feature presented us with the challenge of network topology cycles, in which a message routed from network A to network B via node X might then be routed back from network B to A via node Y, as shown in Figure 8.

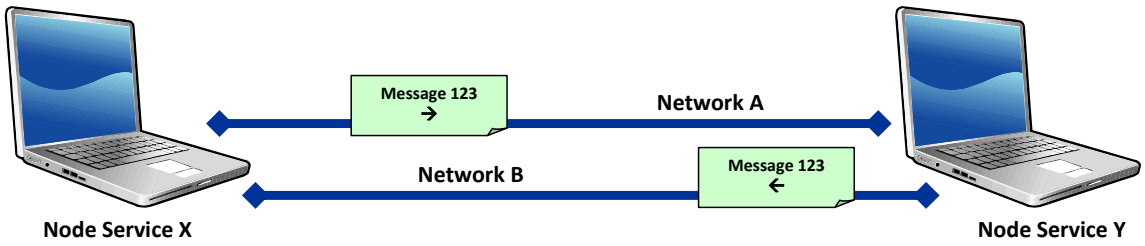


Figure 8: Network Topology Cycle

We avoid excessive relaying by maintaining a cache of recently processed message hashes. If a message block arrives which identifies the hash of an already processed message, then we ignore the newly received block. In addition, the relaying feature will not relay the message to connections for which the node saw traffic for that message previously; for example, if multiple nodes are connected to the same networks. This breaks most message cycles. (Incidentally, a similar hash cache concept is used for the relay of individual message blocks which are being retransmitted in aid of lost packet recovery.)

A hop count is used as a last resort measure to stop message relay cycles; if the maximum hop count for a particular message is 20, then the message will be relayed to nodes which are up to 20 logical network connections away. For multicast connections, the time-to-live (TTL) specification can also be configured for each network connection to limit the number of hops a Gemini packet can take through a routed multicast network. The default multicast TTL is set to 1.

When deploying an Infosphere that spans heterogeneous network segments (i.e. some nodes run on a faster network than others, or the networks are connected via slower networks), then a special "reduced relay" mode is required. Otherwise, the relay functionality will end up forwarding too many control messages and unnecessary block retransmissions. Reduced relay is covered in great detail in section 8.5.1, "Reduced Relay Mode."

5.6.3 Distributed Persistence and Query

While designing Gemini, we recognized that some smaller nodes may not have the capability of persisting data on their own (for instance, they may lack access to a large non-volatile storage area). In addition, nodes may publish and persist information in a limited Infosphere space, then later join into a larger Infosphere, and will need to be able to respond to query requests with results accumulated from the larger Infosphere. We resolved these concerns in two parts: distributed persistence and distributed queries.

5.6.3.1 Persistence Coordination

When a client requests persistence of publications on its publisher sequence, a distributed persistence coordination activity begins. The Gemini Node Service to which the client is connected acts as a persistence "coordinator" for the client's sequence. It looks for and enlists other nodes, which are referred to as "volunteers." Here is the process that takes place:

1. The coordinator announces to the Infosphere that volunteers are requested to aid in redundantly persisting the client's InfoObjects.

2. Upon receiving the volunteer request, other nodes begin to respond with volunteer offers. The responses are sent after random delay periods, and only if volunteers are still needed. This helps to curb excessive network traffic when a large number of nodes are connected.
3. The coordinator acknowledges the first five (by default) responses by sending acceptance messages back out to the Infosphere. The acceptance indicates if further responses are still required.
4. Elected volunteers now ensure that they have subscriptions to the type and version being persisted. They monitor for and will persist those publications which match the persistence sequence identifier.
5. Non-elected volunteers are ignored by the coordinator. The majority of non-elected volunteers won't even send a response back to the coordinator, due to the random delay design. (By the time they get around to responding, they realize that enough volunteers were already elected, so the response is unnecessary and not sent.)
6. When elected volunteers leave the coordinator's Infosphere, then a new election process begins to fill vacant volunteer positions.

5.6.3.2 Distributed Queries

Gemini clients may choose to execute queries against their local node service's persistence store, or against the entire Infosphere. The latter is a distributed query.

1. Distributed queries are sent to the Infosphere.
2. Each node which has a persistence store will then check for matching results to send back.
3. Each node will also pass the query details along to each of their registered query providers (for example, the Mercury bridge, which will execute the query against JBI Mercury's persistence store).
4. Results at a particular node service are then queued for sending back to the Infosphere.
5. The message management component of the node service will prevent most duplicate results from being sent, as the hash value of a result is carefully constructed so that it will match an already cached hash value if the query result contents are identical to a result already received.
6. The node whose client executed the query will now receive the results, further filtering any duplicates that might have slipped through by using its own message hash cache.
7. The results are delivered to the client asynchronously.
8. After a timeout period (the lower of one specified by the client and one configured for the node service), the distributed query is closed to further results and the client is notified of this.

5.6.4 Delivery Modes

Gemini Node Service manages three modes of message delivery: unreliable, reliable, and durable. Unreliable mode may be thought of as a send and forget delivery service. Only a single attempt is made to deliver an unreliable message. Nodes receiving an unreliable message will temporarily cache the message blocks, in case they arrive out of order. But no attempts will be made to recover lost blocks.

With reliable delivery mode, the message is tagged with an expiration period. Until the message expires or is fully received, recovery attempts (as described above in "Lost Information Recovery") will take place.

Unreliable and reliable message delivery modes are accessible by clients publishing via Gemini. There is a third delivery mode, durable, which is currently used for certain internal service messages. Durable messages are reliable with a twist. They are persistent until canceled, and may also be reissued with updates of their contents. Durable messages are periodically retransmitted by the node controlling the message, so late joining nodes still recover them and disconnected nodes eventually remove them from the message cache.

The following table shows the Gemini Node Service message types along with the delivery mode used for each.

Table 5: Message Types

Message Type	Purpose	Delivery Mode
Retransmission Request	recovery of lost message blocks	Unreliable
Sequence Information	announcement of a sequence and its characteristics	Durable
InfoObject Publication	an InfoObject which has been published on a PublisherSequence	Unreliable or Reliable
Persistence Assistance Request	sent by distributed persistence coordinator to enlist volunteers	Durable
Persistence Assistance Offer	sent by distributed persistence volunteer to offer support	Reliable
Persistence Assistance Accepted	sent by distributed persistence coordinator to accept volunteer	Reliable
Query Request	announces execution of a distributed query	Reliable
Query Result	an InfoObject which matches an outstanding query request	Reliable
Node Announcement	announces node presence and includes positive acknowledgements	Reliable, but sent periodically

5.7 Mercury Bridge

We developed a Gemini-Mercury bridge. This application connects to a JBI Mercury Infosphere as well as to a Gemini one. Using the types registered in the Mercury metadata schema repository, it creates publisher and subscriber sequences on both platforms. When publications are received from one platform, they are republished to the other. Using Gemini's query provider interfaces, the Mercury bridge also enables distributed query requests from Gemini clients to be executed against the Mercury InfoObject repository.

5.8 XPath Management

Subscriber and query sequences are always filtered to a specific type and version. A subscription or query execution may be further filtered by an optional XPath predicate. Gemini, like Mercury, applies XPath predicates to a special "combined" view of the metadata components of an InfoObject. InfoObjects currently have two metadata components: extended metadata, which is populated by and in a format specified by the platform, and metadata, which is publisher provided and unmodified by the platform. Here is an example:

Client provided "metadata":

```
<image>
  <name>Sample Image</name>
  <description>This is a sample image</description>
  <priority>1</priority>
</image>
```

Platform "extended metadata," populated once the InfoObject has been published:

```
<og:metadata xmlns:og="http://gdc4s.com/oim/gemini">
  <og:baseObject>
    <og:InfoObjectType>
      <og:Name>test</og:Name>
      <og:MajorVersion>1</og:MajorVersion>
      <og:MinorVersion>0</og:MinorVersion>
    </og:InfoObjectType>
    <og:PublicationTime>2007-12-03T14:30:50.416-07:00</og:PublicationTime>
    <og:InfoObjectID>998c30c4-d333-467d-8c6d-c87acbe6e30c</og:InfoObjectID>
    <og:PublisherID>Client Test</og:PublisherID>
    <og:MetadataLength>121</og:MetadataLength>
    <og:PayloadLength>3269654</og:PayloadLength>
  </og:baseObject>
</og:metadata>
```

In order to evaluate whether an XPath predicate "matches" the InfoObject, we first construct a combined version of the metadata XML components. The client provided metadata becomes the last child element of the platform specific metadata section.

```
<?xml version="1.0" encoding="UTF-8"?>
<og:metadata xmlns:og="http://gdc4s.com/oim/gemini">
  <og:baseObject>
    <og:InfoObjectType>
      <og:Name>test</og:Name>
      <og:MajorVersion>1</og:MajorVersion>
      <og:MinorVersion>0</og:MinorVersion>
    </og:InfoObjectType>
    <og:PublicationTime>2007-12-03T14:30:50.416-07:00</og:PublicationTime>
    <og:InfoObjectID>998c30c4-d333-467d-8c6d-c87acbe6e30c</og:InfoObjectID>
    <og:PublisherID>Client Test</og:PublisherID>
    <og:MetadataLength>121</og:MetadataLength>
    <og:PayloadLength>3269654</og:PayloadLength>
  </og:baseObject>
  <image>
    <name>Sample Image</name>
    <description>This is a sample image</description>
    <priority>1</priority>
  </image>
</og:metadata>
```

Now a subscriber or query execution may match the above InfoObject using predicates such as "`/og:metadata/image`," "`/og:metadata/og:baseObject[og:PayloadLength > 1000000]`," or "`//image[priority = '1']`."

The advantage to the approach we chose is that clients may apply a single XPath which references portions of both the platform and client provided metadata. We did it this way because the CAPI permits a single predicate XML document, and it infers through example that only a single XPath predicate instance is used in this XML document. An alternative would have been for us to define an XML predicate definition which permits use of multiple XPath predicates and allows the writer to dictate whether an InfoObject must match one or all of the provided XPath predicates. This alternative approach

as well as the approach we actually implemented are good in the sense that we do not have to parse the XPath predicate ourselves – we use libraries (built in to Java for subscriptions, and built in to eXist for queries) to do this for us.

Note that by default, "`og`" is recognized as a namespace prefix for the namespace "`http://gdc4s.com/oim/gemini`." We provide this as a convenience for client developers who wish to write XPath predicates which reference elements of our platform specific metadata. An administrator may change the defaults (they are located in a node service configuration file). Client developers may provide a custom list of namespace prefix mappings for individual subscriber and query sequences, overriding the default mappings.

5.9 Type Management

Gemini provides a type repository which clients may choose to utilize. The repository contains entries for each of the descriptors defined by the CAPI: InfoObject descriptors, property descriptors, and attribute descriptors. InfoObject and attribute descriptors pair a type and version to an XML schema. Property descriptors are arbitrary binary data segments which are attached to a specific InfoObject descriptor.

Use of the type repository is optional. We do not see as much value in type management as was seen with Mercury. With Gemini, we anticipate that clients should already know how to correctly form and understand the XML data that they are dealing with. Generally speaking, client developers must be aware, at development time, of the format of the data that they are sending and receiving.

The type repository is useful for clients which utilize the "validate metadata against registered schema" option on publisher, subscriber, or query sequences. It is also useful for clients which want to casually browse through the registered types.

Unfortunately, the type repository is not currently distributed among Gemini nodes. Type registration was not a main focus point in the initial Gemini development effort due to various constraints. Therefore, if someone wants to take advantage of metadata validation, today, he or she must ensure that the types are registered on each individual Gemini node. Fortunately, it is very easy to manually copy the databases from one node to another because they are based upon flat files.

6 Testing and Debugging

6.1 Logging

Gemini Node Service has extensive log outputs. The Gemini installation package installs a shortcut on the Start menu (Windows) or Gnome menu (Linux) to easily tail the log as it is appended. An administrator can adjust the verbosity of logging through options presented while installing Gemini, or afterwards via standard Java logging property files.

System integrators should be able to merge the Gemini logging into other locations as needed – Gemini Node Service uses the standard Java logging mechanism, which accepts plug-ins for output locations.

6.2 Graphical Test Client

We developed a graphical test client for Gemini, which ships with the product. The test client was vital for us during development, as it is an extremely easy way for us to check the operation of the server from end to end. For a detailed description of the Test Client, please see section 5.1.1.1.

6.3 Network Protocol Dissection

Gemini uses a custom protocol for reliable communication over UDP multicast. Debugging the protocol is difficult in a typical Java debugger because once the program or some threads are paused, the timing of conditions on other machines in the network is affected and results vary. So we developed a network "dissector" for Wireshark, which is a zero-cost and open source network packet sniffing tool.

The Gemini dissector is a small C plug-in which Wireshark uses to detect and dissect packets which appear to conform to the Gemini protocol. As packets are captured by the computer running Wireshark, they are displayed in a table and may be dissected further by selecting them, as shown in Figure 9.

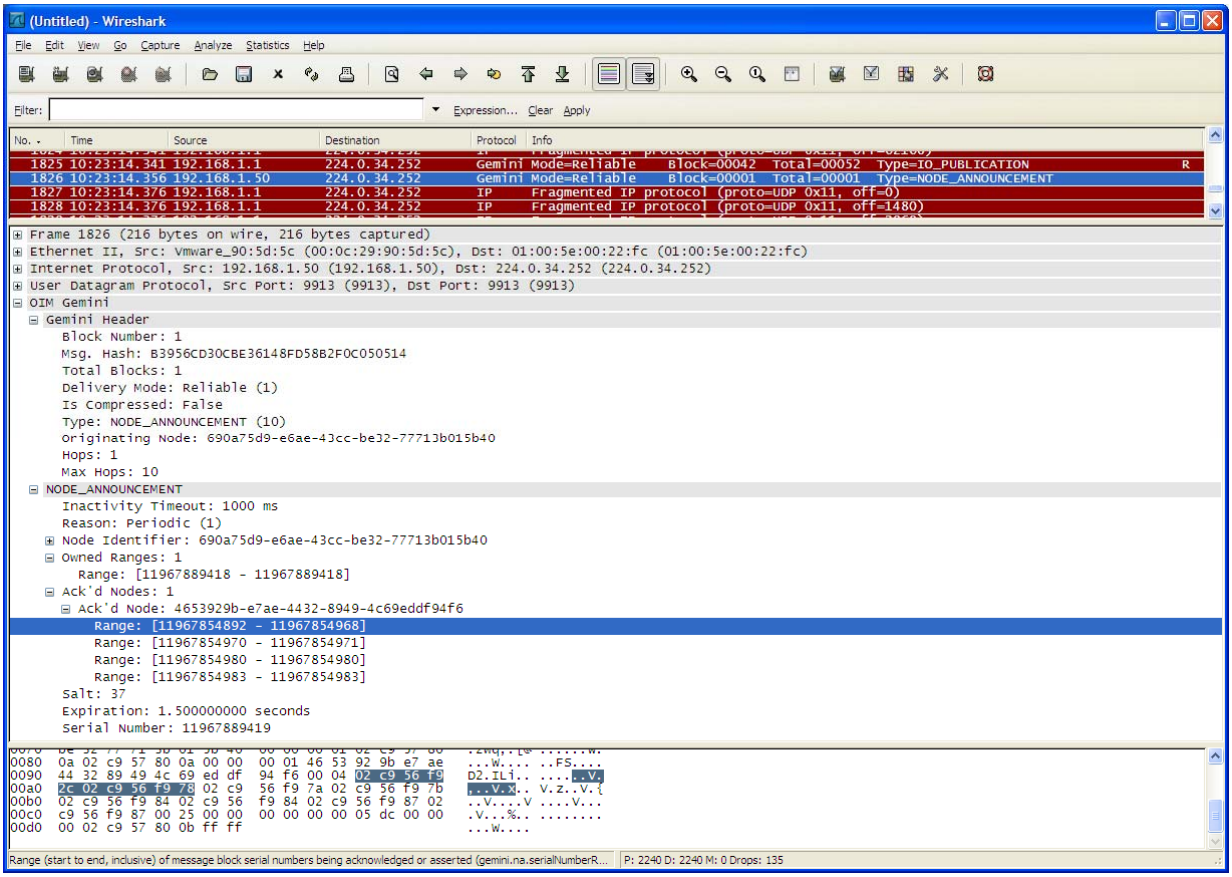


Figure 9: Gemini Dissector in Wireshark

The dissector has proven invaluable for debugging problems with the network protocol. It helped us quickly narrow down whether a problem was on the sending side or the receiving side. It also helped us see the network conversation between connected machines, which helped while trying to reduce excessive occurrences of retransmissions or a lack of retransmissions when they were expected.

The dissector is only partially implemented. Only those pieces that we actively needed for debugging were actually implemented. The dissector identifies (via heuristics) all Gemini messages and decodes the Gemini header on all packets. Only the message bodies of retransmission requests, node announcements, and sequence information are fully dissected, and only for single-block, uncompressed contents.

6.4 Network Emulation

Linux kernel version 2.6 provides a kernel module package known as netem (network emulation) which allows the simulation of various network conditions. In addition, Linux also provides many QoS options that can be set to control how the bandwidth is used on network interfaces. We used a standard PC with multiple network interfaces running the Fedora 6 distribution for sophisticated simulations of disadvantaged networks.

6.4.1 Ethernet Bridge

In most of our simulations, we configured our Linux network emulator to run as an Ethernet bridge. On either side of the bridge, we connected a gigabit Ethernet switch. On either gigabit switch, we connected

our Gemini nodes. In this configuration, the network configuration acts as a single Ethernet segment, as if all the nodes were connected to a single switch. Figure 10 illustrates this network configuration.

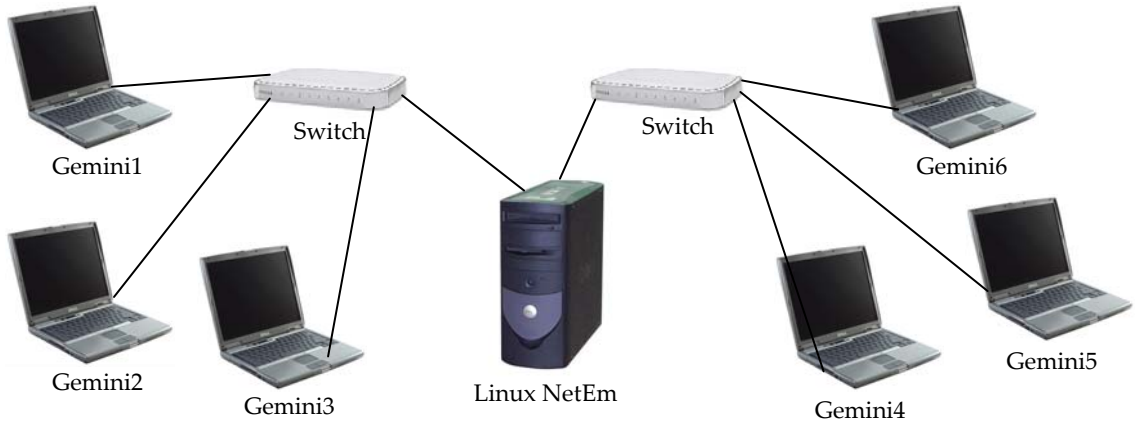


Figure 10: Linux Ethernet Bridge

To test bandwidth limiting or packet loss, reordering, delay, duplication, and bandwidth limiting, we simply configured the appropriate parameters into Linux netem and QoS. Laptops communicating from one switch to the other then experienced the configured degradation. This made it easy to simulate lossy radio connections and the like, to ensure that our Gemini network protocol would overcome the loss, delay, and so forth.

6.4.2 Router

We also configured the same Linux machine to act as a router. We only configured unicast routing; multicast routing was disabled in order simulate many wide area networks where multicast is not routed. We used this configuration, among others, to test our relaying functionality. Essentially, the laptops on each individual subnet all communicated with each other via multicast; one laptop on each subnet was chosen to create a unicast connection through the router, effectively bridging the Infosphere. Finally, as in the case of the Ethernet bridge, we were able to simulate various networking conditions. Figure 11 illustrates this network configuration.

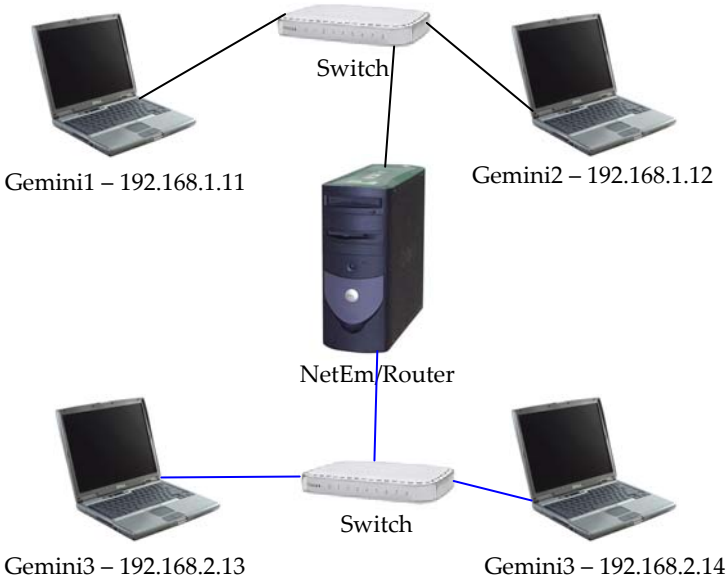


Figure 11: Linux Router

To test bandwidth limiting or packet loss, reordering, delay, or duplication, we simply configured the appropriate parameters into netem. Laptops communicating from one switch to the other then experienced the configured degradation. This made it easy to simulate lossy radio connections and the like, to ensure that our Gemini network protocol would overcome the loss, delay, and so forth.

6.5 Wireless

We tested Gemini using three laptops running different combinations of Windows XP and Linux, with 802.11 a/g wireless cards installed. We tested these running in 802.11 ad-hoc mode as well as separate tests using a wireless access point. We also ensured that Gemini is resilient to having its host computer go out of range of the rest of the network and then rejoining later. An out of range subscriber will receive reliably published InfoObjects when the subscriber rejoins, as long as the subscriber rejoins within the reliable timeout period of the published object.

On Windows XP, if a wireless node loses connectivity to the wireless network (can't see the access point or, in ad-hoc mode, can't see any other peers), Windows will disable the networking stack, affecting the Gemini software (the clients will not be able to communicate with the node service over CORBA IIOP). The best solution when running wireless on Windows XP is to install the Microsoft loopback driver and give it a bogus IP address. Then, when Windows loses wireless connectivity, the network stack remains operational. This problem is not observed under Linux.

6.6 PPP

We established a Point to Point Protocol (PPP) link between two laptops using a serial cable with a configured bandwidth of 19,200 kilobits per second (2.34 kilobytes per second). This allowed us to ensure that Gemini can operate over slower, point to point links such as many of the legacy data capable radios currently in use. We tested this configuration with UDP unicast and multicast, using a Linux machine to act as the PPP server and a Windows XP machine to act as the PPP client. We also tested Linux to Linux using multicast over PPP and Windows to Windows using unicast over PPP. We were unable to get Windows XP to transmit multicast over PPP when acting as a PPP server. Multicast is convenient over PPP as it requires less configuration for Gemini; unicast requires a destination address to be configured in the network configuration file. In our Windows to Linux testing, we also configured the Linux machine to have a second network interface, gigabit Ethernet, to verify that Gemini's application level routing (data relaying) was working correctly.

Early in testing, we discovered problems with our UDP plug-in's configuration wherein data sent to the PPP interface from the Linux machine was being thrown away by the operating system once the underlying PPP interface buffer was full. Gemini reliable message recovery kicks in and eventually recovers the lost packets, but this is inefficient (wastes a large amount of bandwidth) compared to fixing the underlying problem. Research on the Internet revealed that many operating systems do not guarantee that a UDP datagram will actually make it out onto the wire, even when blocking on a socket write (to not overflow the socket buffer); Linux is one of these operating systems. Solaris and Windows, however, do guarantee that the packet will at least make it onto the wire. After experimentation, we found that by setting the socket send buffer size on Linux to the MTU of the networking device or the Gemini maximum message size, whichever is greater, that we were able to get Linux to send out every packet. However, this, of course, has performance penalties, especially added to the performance penalties of the Java networking infrastructure in general. So, on high-speed Ethernet networks, we found that a good trade-off is to set the send socket buffer to around 5 – 10 times the maximum message size or MTU.

6.7 Maturity

Gemini's focus is publishing and subscribing information over decentralized, unreliable networks. Therefore, our testing efforts were concentrated to the client library and client management, message management, network management, and the UDP network plug-in. These areas were tested continuously during development and during preparation for the final demonstration.

In some cases, we went back and rewrote significant portions of code in order to get certain features working properly. The UDP network plug-in, for instance, was significantly rewritten late during project development. The original implementation worked for general test environment situations, but failed in more complex deployments. For example, the plug-in relied on the Java `MulticastSocket` class, which required us to use two different sockets for reading and writing so that we could designate a specific network interface; however, this made it impossible to reuse the port number on multiple sockets. We rewrote this plug-in to use the Java "New IO" `DatagramChannel` even though `DatagramChannel` doesn't directly support multicast; we accessed Java internals through reflection to achieve this.

We also went through a period of extensive network testing with multiple nodes and different configurations. Throughout this testing, we constantly modified and tuned the code and algorithms in order to get the messaging to work as well as possible within given time constraints.

The InfoObject persistence (to eXist) and type management (to SQLite) are not as well tested. For these functional areas, we ensured that the subsystems were functional and worked as expected, but did no stress testing. They are less complex in features and implementation because they were not the main areas of focus during Gemini.

We focused the least amount of effort on security. Although components within Gemini Node Service call into the security component for authentication and authorization, the provided security plug-in is a stub which simply authorizes all actions. When security is implemented as part of some future enhancement to Gemini, the code will require much scrutiny and testing to ensure that security is implemented adequately. Depending on the level of security required, even more scrutiny will need to take place to protect against attacks (denial of service based, false network packets, data access via well crafted XPaths, etc.).

7 Findings

7.1 Lessons Learned

7.1.1 Where Do We Fall Short?

Gemini Node Service is optimized to the concepts of an Infosphere (where information is shared and point-to-point connections are not explicitly created) and for single network connections which support the UDP multicast protocol. If multicast is not utilized, or when multiple network interfaces are used, then the node service will perform tasks which are wasteful. Specifically:

- A client cannot request that a publication be kept local to the node.
- A publication is transmitted on the wire even if there are no subscribers within the Infosphere for that type. While this is generally a wasteful activity, it does permit the existence of silent, receive only subscriber nodes.
- Likewise, a publication will be relayed across network interfaces even when there are no subscribers. This is wasteful for unreliable mode publications, but for those sent reliably, it permits the remote node services to cache the publication. Gemini nodes create a sort of Infosphere-wide cache of unexpired, reliable messages so that the original sender may drop out (temporarily or permanently) and the message will still reach a subscriber who had been dropped out but then reconnected. This is helpful for mobile deployments, although it comes at the expense of additional CPU and memory burden for the remote nodes. Gemini doesn't currently have a delivery mode option for clients to request that only the original sending node provide aid in lost packet recovery.
- Message block retransmissions are relayed across all network connections. In some cases, a block may have been missed by only one node. The negative acknowledgement algorithm on each node listens to the retransmission requests from other nodes, and avoids requesting a missed block when it detects that another node has already made the request. Therefore, the node providing recovery services sends the retransmission to all nodes, just in case. Furthermore, the node service does not maintain any sort of routing tables (e.g. spanning trees) – this was beyond our scope and is rather complex for mobile networks whose connections may change rapidly (see "MANET is not Our Challenge" on page 8).

Our client interfaces are also lacking in some areas:

- Clients may publish InfoObjects unreliably or reliably, but they do not have access to use durability. If we provided access, this would suggest that InfoObjects become mutable. This concept is currently unsupported by CAPI. We did not investigate this further during Gemini, but have filed the concept of client message durability as an enhancement idea.
- Client publishers specify reliability and persistence quality of service attributes. It would be nice if subscribers and query sequences would also request these service qualities.

7.2 Recommended Future Enhancements

Throughout the development process, we documented many enhancement ideas that we knew we would be unable to implement due to time and funding constraints. In addition, we documented other possible enhancements during testing that could improve the robustness, efficiency, and completeness of the current design. These enhancement ideas are documented here and separated by functional area; the listed ideas certainly do not represent all the possible enhancements that can be applied.

7.2.1 General Enhancements

7.2.1.1 Limit Data Structure Sizes

In several places within the Gemini code, data structures do not have adequate protection from growing too large under heavy load and thus could cause the node service to stop functioning. The software should be enhanced by placing configurable limits on all collections, caches, etc. to prevent out of memory conditions.

7.2.1.2 Platform Error Notification

To help system administrators, especially in an enterprise-type deployment, we could add error notifications to the Windows event logging mechanism as well as the Linux syslog logging mechanism.

7.2.1.3 Profiling and Optimization

Profiling the Gemini node service code has not been performed to find the locations of the most inefficient code. We suspect that there are areas within the node service code that could be optimized, without major design changes, in order to gain significant performance improvement.

7.2.2 Client Management Enhancements

7.2.2.1 Support Clients Using Older Java Versions

Not every potential Gemini client may be using the latest Java technologies, so Gemini should build Java 1.4 and 5.0 compatible versions of the Java client library. Since the Gemini client library is very straightforward and does not use many of the newer Java language elements, it should be relatively easy to port the library to support Java 1.4 and 5.0. (JacORB, used heavily in the client library, supports Java 1.4 and up.)

7.2.2.2 Client Libraries for Other Languages

Many applications are obviously not written in Java. Although it is currently possible to write clients in other languages using the CORBA IDL interfaces directly, it is not very practical to expect other developers to fully understand how to do this. Thus, client libraries for other languages, especially for C# (Microsoft .NET) and C++, should be developed.

7.2.2.3 Web Service

A web service client interface has recently been added to Mercury. The web service is implemented externally to the core Mercury software components and uses only the Mercury client library. This is advantageous as it means that Mercury's web service client interface should be easily portable to Gemini.

7.2.2.4 Support Client Connections on Multiple Network Interfaces

If the Gemini node service is configured to accept client connections from remote machines, it can currently only accept the connections on a single network interface. In an enterprise deployment, it is common to have a multi-homed server and thus it makes sense that the Gemini node service should be able to support client connections arriving on each network interface.

7.2.2.5 Queue Data to Clients

Currently, InfoObjects sent to clients based on subscriptions and queries is sent sequentially without going through a queue. Thus, a slow responding client will slow down all other clients connected to the node service. If a client "hangs" in the receive call, then the other clients will not receive data until the client times out or is terminated. Thus, it makes sense to at least add a queue within the client library so that the receive call to the client application is handled asynchronously without affecting other clients. In addition, it may also make sense to add another level of queuing within the node service itself so that the node service is "protected" if someone codes to the CORBA interface directly.

7.2.2.6 Client Query Node Service Details

It may make sense to provide a way for normal clients to inspect a variety of details regarding the server's configuration, such as whether compression is permitted, getting the node identifier, etc.

7.2.2.7 Improve CORBA IDL Interface

Optimize the CORBA IDL for CORBA instead of keeping as close as possible to the Java CAPI; for example, making exceptions in the IDL more granular than that in the Java CAPI, especially since CORBA does not support exception inheritance.

7.2.2.8 Expose Durable Messages to Clients

Many applications need to publish the current "state" of an object while consumer applications may be always interested in what the current state is. Gemini durable messages are a perfect fit for this scenario. Thus, Gemini should expose the concept of durable messages to clients. Normally, InfoObjects are immutable, but exposing durability would change this. When an application starts up and subscribes to data, it would automatically receive the latest version of the InfoObject published (if it is published durably). An example where durability would make sense is the publishing of an air track; points on an air track are periodically updated while the air track is conceptually still the same object.

7.2.2.9 Automatically Reestablish Clients

If a client gets disconnected to the node service, the client library should automatically try to reconnect to the node service and reestablish all existing sequences with the service. This would be useful in an enterprise deployment, for example, when stopping and restarting the node service (maybe to perform an upgrade or fix a malfunction) or if a remote client temporarily loses connection due to a router issue. This can also be mixed with automatically discovering a node service (next enhancement).

7.2.2.10 Automatically Discover Node Service

In an enterprise type deployment, the client library should be able to support automatically discovering a node service from a pool of announced servers in order to minimize the need for configuration, especially if the machine hosting the node service changes. In addition, this enhancement would help to seamlessly support the idea of redundancy and failover; if multiple node services are running on a network and the node service that a client is connected to fails, the client library could automatically establish a connection to a new node service and reestablish all client sequences.

7.2.2.11 Specify Namespace Prefixes by XML

Currently, in order to specify a mapping of namespace prefixes, a client must use a Gemini extension. We overlooked the possibility that the namespace prefixes could also be set using XML through the CAPI sequence attribute mechanism. Adding this capability would make setting namespace prefixes more CAPI compliant.

7.2.2.12 Remove XMLBeans Dependencies in Client Library

Although including XMLBeans bindings to the various Gemini schemas in the client library can be useful to clients, it can cause conflicts if an application already uses a different version of XMLBeans. Thus, the XMLBeans bindings should be moved to a separate, optional library so that clients can still have access to the bindings, but also providing a solution path if there is a conflict with XMLBeans.

7.2.2.13 Subscriber/Querier Reliability

A client may publish data unreliably; however, to a remote client, that data may be important and that client may want to ensure that the data is actually delivered. Thus, it makes sense to also allow subscribers to specify that all InfoObjects on a type and version should be published reliably, regardless of the publisher setting. In addition, query results are sent to a querier unreliably and thus a querier should also be able to request the results be delivered reliably.

7.2.2.14 Incremental Send/Receive

Currently, an InfoObject is sent from the client to the node service in one chunk and then stored in memory. If the InfoObject is extremely large, on the order of many megabytes (like a SAR image), then this is problematic as most nodes would not have enough memory to store the entire InfoObject. Thus, a way should be added for a client to incrementally send and receive pieces of large InfoObjects that can be read from and written to disk directly without using all the system RAM.

7.2.3 Message Management and Networking Enhancements

Message management and networking are very tightly coupled; detailed changes to one component likely ripple to the other component. Thus, the enhancements to both components are grouped together.

7.2.3.1 Reduce Relay Duplication

When multiple Gemini nodes are connected to the same multiple networks (the nodes are relaying to the same networks), the nodes relay the same messages so that the messages are transmitted multiple times onto the relayed networks. So, when relaying messages to other networks, Gemini should prevent the duplication of message transmission when other nodes are also relays for the same networks (possibly by using the same random delay idea that retransmissions use).

7.2.3.2 Reliability to Only Connected Nodes

Gemini could add a new reliability mode which ensures delivery to only the connected, visible nodes at the time the message is transmitted. This reliable mode would not cache messages for longer than is required to ensure delivery to nodes which are online at the time of initial routing. This could possibly free publishers from having to specify a timeout and could greatly reduce memory usage when very large InfoObjects are published. However, this mode would not support delivery to late-joining nodes,

unlike the current reliable delivery mode. This mode could be specified via the reliable sequence attribute.

7.2.3.3 Reliability Where Only Sender Retransmits

Gemini could add a new reliability mode in which only the sender is capable of responding to retransmission requests and positive acknowledgments. This mode would reduce memory usage on other nodes and help with scalability, especially when the combined publishing rate and size of all the nodes is very high. Reduced relay nodes, however, would still need to cache the messages that have been received in order to successfully relay the messages reliably. This reliability mode could also be specified via the reliable sequence attribute.

7.2.3.4 Directed Messaging

Gemini could add directed messaging so that messages could be addressed to just a single node or possibly even a single client. This would be helpful for query results as they are not needed by other nodes. This could also be helpful in supporting a client that only responds to stimulus from another client.

7.2.3.5 Do Not Request Retransmissions When No Subscribers

It may be possible to make a change such that a node does not request retransmissions for lost message blocks when they are known to be for an InfoObject publication for which the node has no subscribers. This would need to be a configurable setting as it would reduce the effectiveness of cooperative recovery. However, this could help reduce the load of a disadvantaged node, such as a node with less memory or bandwidth than other nodes.

7.2.3.6 Add Attributes to Messages for Administration

Some sequence attributes, such as compression and namespace prefix mappings, are not coded into the messages that are transmitted to other nodes. It may make sense to eventually add attributes into the transmitted messages so that more information can be made available via the administration utility and so that more advanced network message routing capabilities may make use of the extra information.

7.2.3.7 Disk-based Caching

The node service could add the use of disk-based caching so that the reliable message cache may hold more messages than what is available from RAM alone. One possibility could be that an InfoObject is cached in memory until sent to all actively visible nodes and once received, the object is then cached onto disk until the message timeout in order to support receipt by late-joining nodes.

7.2.3.8 Optimize Message Management

The current implementation of messaging requires a high amount of processing power in processing reliable messages as well as recovering lost message blocks. This code should be better optimized. It could make sense to write this as native code instead of in Java in order to obtain the best possible performance.

7.2.3.9 Automatic Discovery of Parameters

Gemini timing parameters can currently be tedious to tune when deployed on complex, disadvantaged networks. It may be possible to automatically configure message management routing and recovery delay values based on send and receive timing metrics. This would cover frequency of node announcements, delays and random delay ranges involved with message recovery and relay, timeouts for unreliable message recomposition, and so on.

7.2.3.10 Versioning

Currently, if additions or changes are made to the Gemini packet headers or structure, then that version of Gemini will be unable to communicate with an older version of Gemini. This is not a problem today since Gemini is not deployed, but it could eventually become a problem if Gemini gains widespread usage. Thus, implement versioning into Gemini's data structures, messages, and network packets could allow future versions of the node service to interoperate with an older version.

7.2.3.11 Other Compression

Gemini currently allows gzip compression to compress both binary and XML data included within an InfoObject. Although gzip certainly makes a large improvement in total InfoObject size when an InfoObject contains a large amount of XML, it is not nearly as efficient as a product called *Efficient XML* by AgileDelta (see http://www.agiledelta.com/product_efx.html). In addition, there are other better compression techniques than gzip, such as bzip2, that may be able to compress the binary data more effectively. Together, by optimizing both XML and binary compression, Gemini could make more efficient use of low bandwidth links.

7.2.3.12 Additional Information in Platform Metadata

InfoObjects should contain additional information in the platform metadata area. For example, the metadata could include both the sending node's UUID as well as a list of the sending node's IP addresses to help an administrator determine where a message is coming from in case a client is misbehaving. (The client connection name may not be enough on a large network.)

7.2.3.13 Flow Control

Gemini does not perform any flow control over its reliable UDP protocol. This can be a problem when disparate network speeds and/or processor speeds are in use, as exemplified by the following scenarios:

- Nodes A, B, C, and D are all connected to a gigabit Ethernet switch; however, node D only supports 100 megabit Ethernet and is thus connected at a slower speed. If node A transmits a large volume of data, nodes B and C will have no trouble receiving the data, initially. However, node D will experience a large loss of packets simply because of the slower speed that the node is connected at. This will cause node D to send out a large volume of retransmission requests and will also increase the size of its node announcement message due to the positive acknowledgment list getting fragmented. This, in turn, causes a large processing burden on the other nodes, which could lead to nodes B and C to lose packets because the processor is unable to keep up. This then causes nodes B, C, and D to all lose different packets, causing even more retransmissions. Eventually, the network of Gemini nodes starts to thrash and recovery becomes improbable.

- Node A is on a gigabit Ethernet network and node B is also on a gigabit Ethernet network; however, these nodes are located remotely from one another and all traffic must pass through routers and a WAN that is in between. The effective bandwidth of the route may be on the order of 1 megabit. If node A transmits a large volume of data, node B will lose most of the packets that were transmitted, as in the previous scenario. The messages sent will eventually be recovered, but node A will transmit far many more packets than actually make it through the route. This has the negative effect of saturating the router to where ***other***, non-Gemini, network communication has trouble getting through. The router could also be replaced by a slow radio that connects via Ethernet; the same problem arises.
- Clients on a node together publish a huge volume of data at a high bandwidth that the underlying Gemini platform and network connection is incapable of supporting. Since publications are cached, the node service memory usage continually grows until exhausted.
- Other nodes on the network publish too much data volume for the local node's resources. The local node will lose many packets and the message cache could grow to exhaustion.

There are essentially two types of problems here. First, clients are allowed to publish at an unlimited bandwidth to the node service. Second, nodes can publish to each other at an unlimited rate. To address these issues:

1. The node service's UDP plug-in could allow a bandwidth limit to be set such that the plug-in will never transmit over that specified limit. This flow control would need to flow upwards so that the clients themselves are blocked from sending when the network reaches this limit and when the queues are full.
2. When nodes on the network discover that there are an excessive amount of retransmissions, then the other nodes should back off on the bandwidth of their transmissions, because, most likely, the excessive retransmissions are caused by flooding the network or a node's processor. Again, this flow control would need to flow upwards to block clients from sending when the buffers become full.
3. Implement a transmission window for each node so that there are only a certain number of active message blocks on the wire from each node at a time. This would be similar in concept to the windowing that TCP uses. Again, if the queues are full, flow control would need to flow upwards to the clients.

7.2.3.14 Interfaces Specified by OS Name

Due to Java limitations, the network interfaces in Gemini are specified in the configuration by IP Address instead of by interface name. This presents a problem when an administrator wants to deploy a standard configuration to a wide number of nodes where a specific interface is used. The configuration needs to be slightly modified on each node so that the IP address of the interface can be specified. If the interface is specified by operating system name instead; for example, "Local Area Connection" on Windows, or "eth0" on Linux; then the configuration can be more easily deployed. This especially comes into play when imaging machines with tools like Ghost and Alteris. Specifying the interface by operating system requires direct access to the operating system; requiring native code such as C++.

7.2.3.15 Native Networking

We experienced many problems with Java and multicasting and attempted to work around them as best as possible. Java multicasting simply does not sufficiently provide the advanced features needed to implement a complex, multicast-based product like Gemini. Specifically, but not limited to:

1. Java 6 multicasting has many bugs and shows different behavior between Linux and Windows.
2. Java 6 requires binding to a UDP port on all interfaces in order to receive multicast data on just a single interface; this disallows using the same port number for multiple multicast and unicast connections.
3. Java 6 does not provide a way of getting to the destination address of UDP datagrams to verify that it is a packet that we should process.
4. Java 6 does not officially support Multicast DatagramChannels (this is "hacked" on Gemini; only Sun's JVM is supported as Sun internals are used as part of the hack) – Official multicast support is not planned until JDK 7.
5. Java 6 network performance is slow and appears to have a high CPU impact.
6. Java 6 does not provide access to many advanced operating system features, such as obtaining all the interfaces on the system by their friendly name (Windows).
7. Java 6 does not support other networking protocols, such as SCTP.

Rewriting the Gemini networking code in C++ could address many of these issues.

7.2.3.16 Utilize Other Low-Level Protocols

Other lower-level protocols may be useful to Gemini. Creating a TCP network plug-in may allow Gemini relaying through a router to work more effectively with little additional software work. TCP also allows easier firewall configuration (when NAT, Network Address Translation, is utilized).

SCTP, which is essentially a cross between TCP and UDP, also looks promising for both Gemini and JBI in general. SCTP can be reliable like TCP, but is message oriented like UDP. SCTP can support multiple, separate *streams* of data on one socket, unlike both TCP and UDP. Finally, and most important, SCTP was designed to work with multi-homed hosts, automatically transmitting data to the correct interfaces. Unfortunately, SCTP is still not widely adopted, and thus, is not supported on Windows by Microsoft; however, it is supported on both Linux and Solaris. Please see http://www.linuxjournal.com/article/9748, http://www.linuxjournal.com/article/9749, and http://www.linuxjournal.com/article/9784 for a good in-depth overview of SCTP.

7.2.3.17 Hierarchal/Intelligent Relaying

To address Gemini scalability, the Gemini protocol could be modified to support a node hierarchy. This is best described using the figure below:

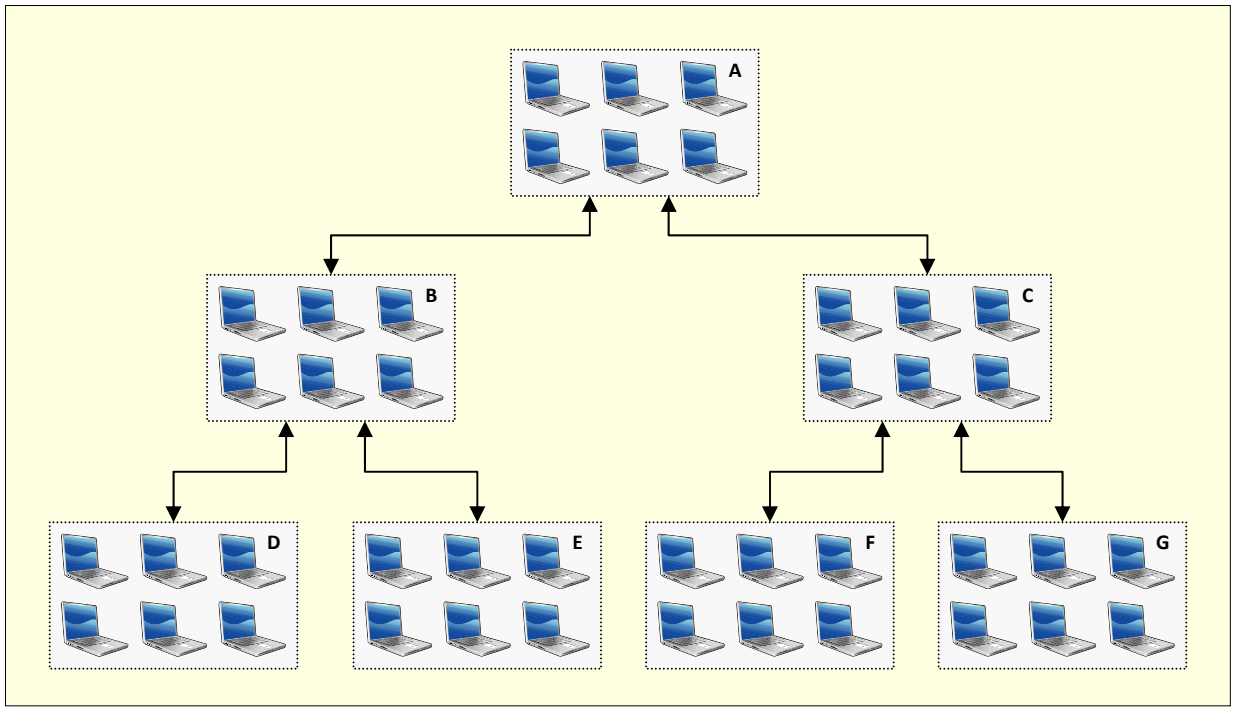


Figure 12: Node Hierarchy

In this configuration, the nodes in each group would communicate like they currently would today. However, the nodes in each group would only directly receive messages within their group as well as from the group below them (for instance, group B would receive messages from nodes in groups B, D, and E). Each higher level group of nodes would be responsible for relaying the message to other groups, if appropriate (there are subscribers, group has permission, etc). Gemini control messages would not pass from one group to another, except to the next higher group (for example, group E would not receive node announcements, retransmission requests, etc. from group D; group A would not receive the same type messages from groups D, E, F, and G).

This discussion just scratches the surface of how hierarchies in Gemini would work as the design for this would require much thought and preparation. However, a hierarchal relationship can actually be manually configured today using reduced relays and many separate multicast groups. Thus, the possible performance benefits can be studied to help ensure a good design.

7.2.3.18 Burst Transmissions

Legacy radios require time to initiate and close a transmission, on the order of hundreds of milliseconds. In order to improve transmission efficiency, it may make sense to allow a network connection to be configured to queue data up to a certain amount of time so that the data can be transmitted all as one burst, thereby reducing the number of times the radio is "keyed."

7.2.4 Administration Enhancements

7.2.4.1 Complete Administration Utility

The administration utility is currently a shell for future development. It provides enough of a framework to perform certain operations, but it is mostly unfinished. The utility should be completed in order to allow an administrator to see, gather statistics, and snoop on all nodes, publishers, subscribers, and queriers on the entire network. In addition, the utility should allow for a robust view and management of the type repository as well as the stored InfoObjects. The utility should be able to list and manage authorized users as well as provide a graphical means for an operator to adjust Gemini parameters.

7.2.4.2 Remote Node Administration via the Gemini Protocol

Currently, in order to administer a remote server, you must create a client connection directly to that server. For a large network of Gemini node, it makes sense to develop a mechanism so that remote nodes can be administered via the Gemini networking protocol, allowing an administrator to see and configure all nodes at once.

7.2.4.3 Administration Web Page

For an enterprise type deployment, it may make sense exposing the administration capabilities via web pages instead of just using a client application. This would not make sense in a distributed environment.

7.2.4.4 Distributed InfoObject Destruction

Today, when an InfoObject is destroyed, it is only destroyed on the local node; specifically, it is only removed out of persistence on the local node. When a client requests that an InfoObject be destroyed, it should be destroyed on all nodes.

7.2.5 Subscription Management Enhancements

7.2.5.1 In-Order Delivery

Subscription management could be modified in order to support the in-order delivery of received InfoObjects with regard to a publisher.

7.2.5.2 No Publications from Own Client

A sequence attribute could be added so that a subscriber could specify that it does not want to receive publications that are published on the same connection. This makes sense for most applications as they do not need to receive what they publish.

7.2.5.3 Local-Only Publications/Subscriptions

A sequence attribute could be added so that a publisher could specify that a publication is only intended for the local node and that it should not go out on the wire to other Gemini nodes. Many systems have internal software components that need to communicate and share data all within one computer; it does not make sense to send these publications out to other, uninterested nodes.

7.2.6 Persistence Enhancements

7.2.6.1 Get Most Recent Publication

Provide a way for clients to retrieve the most recent InfoObjects published which satisfy a criteria. For instance, return most recent red/blue forces (where client provides hint to API to specify that the "/metadata/force/[ID]" is their unique ID field).

7.2.7 Type Management Enhancements

7.2.7.1 Distributed Type Management

Provide distributed type management, such as registering a type on all nodes instead of just the local node.

7.2.7.2 Schema Includes and Imports

Support includes and imports in schema definition files, much like the recent capability added to Mercury.

7.2.8 Security Enhancements

7.2.8.1 Encryption

Gemini could encrypt all client and node service communication using the built-in capability of most modern CORBA ORBs to use IIOP (Internet Inter-Orb Protocol) over SSL (Secure Sockets Layer). In addition, all communication between all nodes using the Gemini protocol could also be encrypted, possibly with SSL as well.

7.2.8.2 Authentication

Gemini does not really provide authentication today as discussed earlier in this document; authentication is essentially stubbed for future development. Aspects of authentication on Gemini that need to be developed include:

- Support distributed authentication using the Gemini protocols
- Distributed management of authentication
- Support enterprise authentication, such as LDAP
- Granting and revoking node trust dynamically

7.2.8.3 Denial of Service/Exploits

The Gemini code and protocol should be analyzed and changed to minimize denial of service attacks or exploits that could result in security breaches.

7.2.9 Wireshark Dissector Enhancements

The Wireshark Gemini dissector is unfinished; it does not support the decoding of messages containing multiple message blocks, compressed messages, or some control messages. The plug-in also does not implement statistical analysis and reporting similar to other Wireshark built-in protocol dissectors. In addition, the dissector has not been compiled or tested on Linux.

7.3 Documented Defects

The following table lists open defects as documented within the Gemini defect management tool, ClearQuest. With the exception of two entries, each of these defects is in the *Open* state, meaning that detailed analysis has not yet been performed. Gem-00006 and Gem-00057 are in the *Verifying* state, meaning that work was performed to resolve the defects, but the fixes have not yet been verified.

Table 6: Open Defects

Priority	Number	Description
High	Gem-00057	**Unexpected (unnecessary) recovery retransmissions occur** (Verifying) Update ReliableMessageBlock to track the time that a retransmission is QUEUED versus the time it has finished SENDING. This will account for potentially large delays between the two. Without tracking both, we could end up sending a block into the queue multiple times before it is finally sent. Move the code that is in common between pos and neg. ack management into a common file (Retransmitter). Use the minimum request delay configuration item for pos. acks in the same way as it is currently used on neg. acks. Do not record a transmission time or request time on the message block until one of those events actually occurs. Be much smarter about tracking retransmission request time stamps, such that we don't forward the time stamp unless it is past the completion of a previous request, and so on... See the code; this is complicated to explain here. Update the network code so that the network plug-in provides notification when it has completed an on-the-wire send of a packet (to the level of knowledge which is reasonable for the plug-in to do...).
High	Gem-00060	**Gemini server using 100% CPU and not recovering** While sending multiple large messages reliably (say 1 MB) to a remote Gemini server, sometimes the sending node and/or the receiving node enter into an unrecoverable CPU spin. When this occurs, the server must be terminated and restarted.
High	Gem-00061	**Gemini server losing a lot of packets while receiving large reliable messages** Sending large reliable messages causes Gemini server to process a lot of data. This seems to cause the server to lose packets as they are not being read off of the socket fast enough. This causes both a waste of network bandwidth as well as additional processing overhead to recover the lost packets. The workaround, at least on Windows, is to set a huge socket receive buffer.
High	Gem-00062	**Reliable message recovery occasionally fails while servers under heavy load** Test setup: VMware XP machine running inside of real XP machine. Both machines running server and test client. VMware machine is publishing 50 one megabyte InfoObjects with 0 ms delay, and 90 second reliability expiration. Publications and almost all of the lost packet recovery occurs within the 90 seconds, but while time is still remaining, recovery seems to stop for zero to three remaining of the original 50 messages. (Often times all 50 make it, but in some tests, 1 or 2 messages are not fully recomposed.) It appears that the server which should be providing recovery blocks (VMware machine) is sitting silently while the server which needs blocks (real machine) is actively requesting them. This is likely a problem in the algorithms used in the class com.gd.c4s.gemini.svr.msg.impl.Retransmitter, or maybe in both PosAckHandler and RetransmissionRequestHandler from the same package.

Priority	Number	Description
High	Gem-00081	**High CPU usage and wasted bandwidth when reliable msgs expire** When sending 100 1MB messages on a single publisher to 5 receivers with a 60 second expiration, the CPU usage and retransmissions become high when the messages start expiring (all the machines already sucessfully received all messages). What appears to be going on is that not all nodes expire a message at the same time. When node A expires the message when node B has not yet and node B gets a node announcement from node A, node B tries retransmitting all the packets for the message to node A (even though node A actually already received the message). As you send more publications, the situation becomes worse to where the CPU usage remains high and the node service never recovers, forcing a restart.
High	Gem-00084	**Retransmission requests showing as malformed in Wireshark** Many retransmission request packets are showing up as malformed within Wireshark. This could either be a problem with the Wireshark plugin itself, or worse, the Gemini node service code itself. If the problem is in the Gemini node service, this is an important defect to fix.
High	Gem-00085	**Gemini Node Service hangs/spins CPU under load with many retransmissions** In the lab with 6 Gemini nodes, 5 Windows machines and 1 Linux machine. If you transmit 2000 10K messages from all nodes simultaneously (with each node also a subscriber), then the Linux machine causes many retransmissions to occur (separate performance defect). Usually at least one node will eventually hang spinning the CPU. The only recovery is to kill the node service and restart it.
High	Gem-00086	**Node service performance degrades over time** If you publish a good quantity of messages between many machines while many retransmissions occur, then the publishing/receiving of messages becomes slower and slower over time. For example, if I have 5 nodes publish 2000 messages every 10ms simultaneously and have a decent amount of packet loss to cause a lot of retransmission requests, then the first I transmit all nodes may receive the data say within 50 seconds, the next time maybe 90 seconds, the next time 3 minutes, etc... This could be related to the hash caches getting larger. Maybe we need a way to age the hash cash entries so they expire after an amount of time. In addition, maybe we should drop retransmission requests having a hash, if possible.
Medium	Gem-00006	**Node ID is not persisting w/ non-admin account** (Verifying) The Gemini server node ID is not persisting correctly when running as a regular Linux user. Problem may exist on Windows as well (not tested). This causes a new node ID to be generated each time the Gemini server is started, which is not our default, desired behavior.

Priority	Number	Description
Medium	Gem-00051	**Recovery retransmissions are not relayed across network connections** While thinking about it, we realized that when message blocks are intentionally retransmitted (for lost block recovery purposes), they are sent from the retransmitting node to its connections, but the connected nodes are not forwarding it any further. They need to forward further since the node requiring the retransmission may be more than one hop away. This problem is not a big deal YET because right now, all nodes cache all the same reliable messages. But it could easily become a problem when a neighboring connection node has a small cache and had to expunge the blocks that are needed to recover someone near by.
Medium	Gem-00087	**Poor Linux performance compared to Windows** When publishing 2000 10K messages every 10ms with 6 nodes simultaneously with one node Linux and the others Windows, the Linux node gets far behind and appears to cause a slew of retransmissions to occur (unverified). Because of the retransmissions, the Linux node effectively slows down all the Windows nodes to where the publishing/receiving slows down to a crawl as well as causing high CPU load on all nodes.
Medium	Gem-00088	**Cooperative packet recovery causing more retransmissions than necessary** When there are 3 machines connected to ethernet and another machine connected through the ethernet bridge with packet loss and we send 1000 1 block messages from a machine on ethernet, all 1000 messages are recovered on the lossy side and all machines on the ethernet side helped recovery; however, the more retransmissions occured than what was necessary. This needs to be investigated.
Low	Gem-00009	**Duplicate query results may occur when IO persisted in heterogeneous repositories** Currently, the IOR is supplied with only an eXist database plug-in. If additional XML database plug-ins are implemented in the future, there is a possibility that a distributed query will result in some duplicated InfoObject results being delivered back to the client. This is because the servers utilize a message hash to determine what is or is not a duplicate result. If eXist reformats the InfoObject metadata in any way, or if a different XML database provider does so, then this will cause the resulting QueryResult message object from each to encode out to a different byte array, having a different calculated hash value. One possible solution is to have the server "clean" the XML, cutting it down to the smallest possible size (minimal whitespacing). This isn't very nice, since it means we'll be drastically changing the appearance of the XML our clients provide. And it's not totally foolproof (the database could inject other changes that we didn't plan for). A better solution might be to place the message hash into a new field in the extended metadata. Then, have a new send() method on message manager which the persistence manager would use to forcibly set the message hash instead of calculating it.
Low	Gem-00011	**Duplicate query results may occur with heterogeneous MessageDigest config** Distributed query results may be received more than once by the client that is executing the query. Specifically, this can happen when the following conditions are ALL true: - The InfoObject being returned is present at multiple servers. - At least one of the servers is configured to use a different MessageDigest algorithm than the other servers.

Priority	Number	Description
Low	Gem-00042	**Persistence Manager Synchronization** While implementing the delete methods in persistence, it doesn't appear to me that the persistence manager and the DAO packages have sufficient synchronization to handle multiple simultaneous requests from several threads. The code needs to be analyzed and proper synchronization should be put in place. Since persistence was not high on AFRL's list, the priority of the defect should be low since this is a proof of concept.
Low	Gem-00043	**Persistence Manager not adequate exception handling (forward up the chain)** While implementing the delete methods in persistence, I found that the exception handling to be inadequate in the persistence manager and the DAO packages. Specifically, although exceptions are handled, the error conditions are not passed up the chain so that a client can be informed of the error conditions.
Low	Gem-00047	**Exist plugin memory leak on queries** Through inspection, it appears that the exist plugin never releases the query UUID from the map containing active queries when a query is finished. Thus, the data associated with a query is kept indefinitely.
Low	Gem-00052	**Possible cycles in connection relaying** If Gemini is configured for use with UDP unicast links, and the links are bidirectional in nature and reasonably well connected (i.e., node A sends to B, B to C, C to D, D to C, C to B, and B to A) then we have a problem where data will be cycled needlessly between all of these nodes. Our TTL mechanism (hops and max hops) will stop a cycle, but only after excessive hopping has already taken place. This can be pretty bad for a disadvantaged network or one which is overly well connected. This can be exceptionally bad when it comes to block retransmissions. That is, blocks which are resent to each connection as the result of lost block recovery (the pos/neg ack systems). In this case, it is more difficult for us to remember that a retransmission block has already passed over a given connection route, and so we rely solely on hop count to stop it. Therefore, as is described in Gem-00051, we should purposefully reduce the hop count on retransmission blocks to put a better limit on this problem. A great long term solution will probably require keeping/using routing tables. The block retransmission events described above are now handled in a manner extremely similar to original message routing, whereby hashes of recently retransmitted blocks are maintained to reduce incidence of needless relays across network connections. This particular problem was fixed in Gem-00051 ("Recovery retransmissions are not relayed across network connections").

Priority	Number	Description
Low	Gem-00058	**InfoObject which arrives after pub sequence is terminated may not be persisted** If some node publishes an InfoObject on a persisted sequence, and then immediately terminates the sequence, it is possible that a remote node which is volunteering its persistence store support, may not persist this InfoObject. This would occur if the network is delivering packets out of order and the publisher sequence termination is received before the last publications on that sequence. As a work around for now, client developers may want to leave publisher sequences open until client termination, or at least insert some delay before terminating a publisher sequence such that it does not occur just after the last publication. Note that the persistence store on the server local to the publishing client is unaffected by this, as it should receive the messages in order. Likewise on a network which guarantees delivery order.
Low	Gem-00059	**local query performed after distributed query on same sequence causes null pointer exception** I modified the example client code to execute a distributed query, and then when the query completes, to execute a local query on the same sequence. (Comment out the "else" statement which follows the if (distributed) block in the query() method.) I then run the test client with the following arguments: java -jar lib\example-java-client.jar -query test 1.0 -distributed When the distributed query completes, the standard executeQuery() is called. During this call, a NullPointerException is thrown from ExistDatabaseManager.getTypeVersionString(). TEMPORARY WORKAROUND: Clients should not convert usage from a distributed query to a local query within a single QuerySequence. Perform local queries in a separate query sequence.
Low	Gem-00064	**Mismatched MessageBlockMaxSize may cause corrupted query result** If two nodes are configured with different values for MessageBlockMaxSize (in GeminiMessageManager.properties), and these two nodes are simultaneously returning the same QueryResult message for a distributed query request, then the nodes receiving the query response will corrupt the message as they are receiving it. This will occur because the receiving node first receives a block of the query result message from Node A. Say it is 1000 bytes. Now the receiving node is expecting all further blocks to be 1000 bytes, and is expecting let's say 10 total... But Node B is sending 2000 byte blocks, 5 total... So there is a mismatch. Since the receiving node is not on the lookout for this occurrence, it will corrupt the received message.

8 Deployment

8.1 Appropriate Uses of Gemini

Gemini provides a framework which offers client developers the opportunity to publish and subscribe to information in a many to many style relationship pattern. It provides reliability and data distribution services for operation over disadvantaged networks. It bridges unrelated network segments. It also provides a means to archive and query the published information. Any system whose applications need to share information in a distributed, asynchronous, and decentralized fashion should consider using Gemini.

8.2 Limitations

Gemini is not appropriate for clients which require synchronous communication (wherein a publisher and subscriber are processing some piece of information in lock step). Gemini also does not replace object-oriented communication frameworks, such as CORBA and RMI, where remote objects can be created for use only with that client. It is also not appropriate for clients that have real-time streaming requirements, such as carrying a voice conversation or streaming a live video feed.

In the subsections which follow, we describe various deployment patterns that one may choose from while arranging networks of Gemini clients. It is important to note that we have certain deficiencies with our current node service implementation, which restrict the ability for Gemini to scale in certain ways:

- Reliable messages are cached in RAM. Clients specify the number of seconds before which reliable publications expire. The total number of active, reliable messages that may be cached depends on the available RAM. Client developers need to choose expiration of reliable messages with care.
- Gemini nodes discover each other and recover missed, reliable data packets by periodically sending out node announcements to each other. If a network has low bandwidth (such as a typical military radio network) and has a lot of nodes, then the frequency with which node announcements are published needs to be scaled back so that enough additional bandwidth is available for client data. Reducing the frequency of node announcements increases the latency associated with lost packet recovery.
- Similarly, a node with low processing capability may be overwhelmed by the message management it requires to keep up with the node announcements and other data flowing in from remote nodes. So again in this case, node announcement frequency should be reduced. Increasing the delay of transmitted negative acknowledgements so that messages are more likely to be recovered using positive acknowledgments (located in the node announcement) will help; this reduces the burden of processing the negative acknowledgments on other nodes. Both of these configuration changes increase the latency associated with lost packet recovery.
- When multiple network connections on a single machine are bridged by the Gemini Node Service, the "routing" is not sophisticated. Therefore, when a variety of network segments are involved, it's quite possible that data will relay through Gemini nodes which have no need for the information (e.g. no active subscribers and no pass-through route to another node that does have subscriber relevance). The downside of this is that the node service at each node is forced to process some information that it turns out wasn't even needed. The upside is that this form of Gemini network configuration is optional: if the entire Infosphere is addressable via a single multicast group, such

as should be the case when MANET overlays and mesh networks mature, then the lower level network layers will handle this routing on Gemini's behalf.

- In addition, a relay can easily flood one of the network connections it is connected to; for example, if a node is relaying from gigabit Ethernet to PPP. Gemini does not perform any flow control over its reliable UDP protocol as explained in section 7.2.3.13.
    - As an intermediate workaround, we implemented a "reduced relay" mode which helps in this deployment scenario. See 8.5.1 ("Reduced Relay Mode") for details.
    - Applications must be written to accommodate the network bandwidth on which they will be deployed. They must be tuned to not transmit at a bandwidth that overwhelms the node services on the network or the available network bandwidth.

8.3 Mobile, Ad-Hoc Networks

The deployment of focus is a decentralized, wireless network with mobile nodes. In this case, Gemini Node Service should be deployed to each computer which needs to run Gemini clients. This way, the clients may connect to the server running on their own machine, guaranteeing reliable network connectivity to the server. The node services running on each mobile platform then discover each other whenever they gain network connectivity, causing the clients' information space ("Infosphere") to automatically become larger. A simple example of a tactical deployment is shown in the following figure.

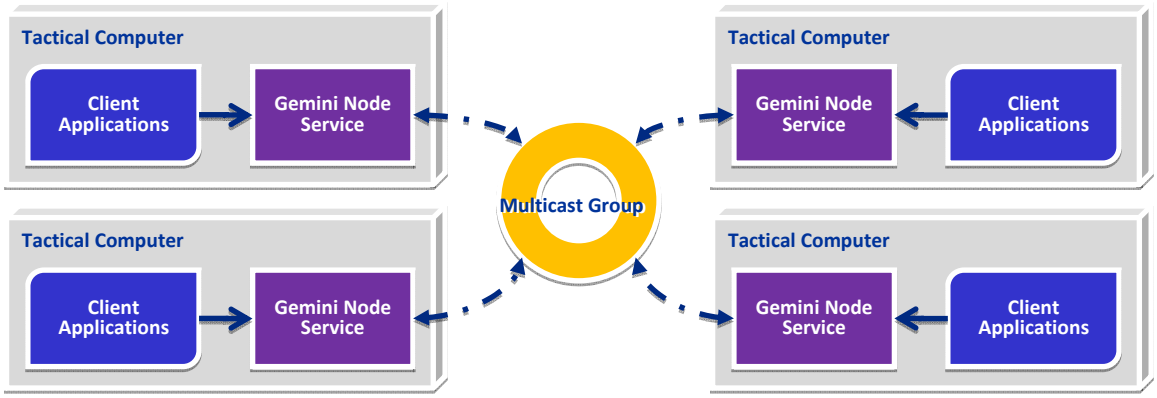


Figure 13: Example Tactical Deployment

An important note for mobile deployments on the Microsoft Windows operating system: We've observed that when we bring an ad-hoc mode wireless node out of range of any peers, Windows removes the wireless interface from the list of enabled network interfaces and disables the networking stack. When the node is back in range with at least one peer, then Windows re-enables the network interface. Gemini Node Service survives the network interface modifications. However, clients connected to the node service use TCP, and their socket is broken when the interface becomes disabled. Therefore, in this deployment configuration, we strongly recommend installing the Microsoft Loopback Adapter, assigning a dummy IP address to it, and configuring Gemini clients to use 127.0.0.1 (the "localhost" address, which is the default) to connect to the node service. This may be configured in the `GeminiClient.properties` and `GeminiClientProxy.properties` files, located in the `data` folder of the Gemini installation.

8.4 Wired Infrastructure ("Enterprise")

When Gemini is deployed to a stable network, such as wired Ethernet in a building or within a private network in a tent, then it is not necessary to run the node service on every client machine. Instead, an

administrator may choose to deploy the node service to a specific machine and have that machine handle connections from multiple remote clients on the wired network. Clients connect to the server via reliable sockets (TCP), as illustrated in the following figure.

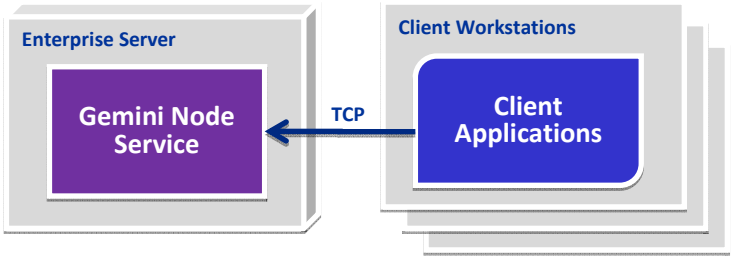


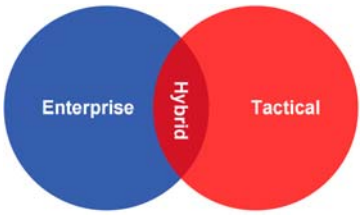
Figure 14: Example Enterprise Deployment

This mode of deployment works exceptionally well. Performance is very high. We saw sustained transfers as high as 60 megabytes per second between clients on two machines that were connecting to the same node service running on one of the machines. In this configuration, the UDP network plug-in can even be disabled (by not configuring any networks within `GeminiNetworkManager.properties`); the highest performance, for a low number of subscribers, can be obtained in this configuration as the publications do not need to be split into message blocks and be multicasted onto the network.

An enterprise based deployment might want to stand up redundant servers to withstand failures in hardware. Gemini supports this by simply running the node service on multiple machines (assuming the node services are configured for multicast). An administrator may configure Gemini to redundantly persist data (by default, this will occur at up to 6 machines). Currently, the Java client library requires an IP address and port to connect to a node service. It could be enhanced such that the node service broadcasts its presence on the wired network, and the client library automatically would choose a node service to connect to, and fail over to another node service if the connection is ever lost.

8.5 Hybrid

Some combination of the above is also acceptable. An example scenario: a group of soldiers, each outfitted with a computer and radio, are executing a mission on foot and sharing information electronically with their command which is back in an operations center tent. In this scenario, each foot soldier would have his Gemini clients and node service running within his computer, while the operators in the tent will share use of just one or a few servers which are collocated with them on their wired network.



Regardless of whether they have connectivity, the foot soldier's node service is able to store important InfoObjects. While the soldiers are in proximity of each other, the node service on each soldier will automatically begin information sharing, enabling their clients to communicate. Once the soldier network includes connectivity to the operations tent, then the soldiers working within it are also able to participate in the information exchange, as well as run queries to retrieve the important, archived InfoObjects from their distributed storage locations.

When Gemini Node Services are distributed across multiple networking environments, such as on different LAN segments which are joined via a WAN, then this too is considered to be a hybrid deployment.

In most cases, hybrid deployments will require a specialized form of network partitioning which we refer to as "reduced relay" mode. In the following subsection, we will go into great detail on reduced relay mode, to show why it is vital for deployments which contain heterogeneous network segments.

8.5.1 Reduced Relay Mode

Consider the deployment shown in Figure 15, below, where the Infosphere consists of Gemini nodes which span heterogeneous network segments. In this example, there are three laptops on a gigabit Ethernet segment and three laptops on a wireless segment. One of the laptops (the one in the center of Figure 15) is connected to both network segments and has Gemini configured with a network plug-in instance for each segment, so that this laptop will provide relay bridging services between the two segments.

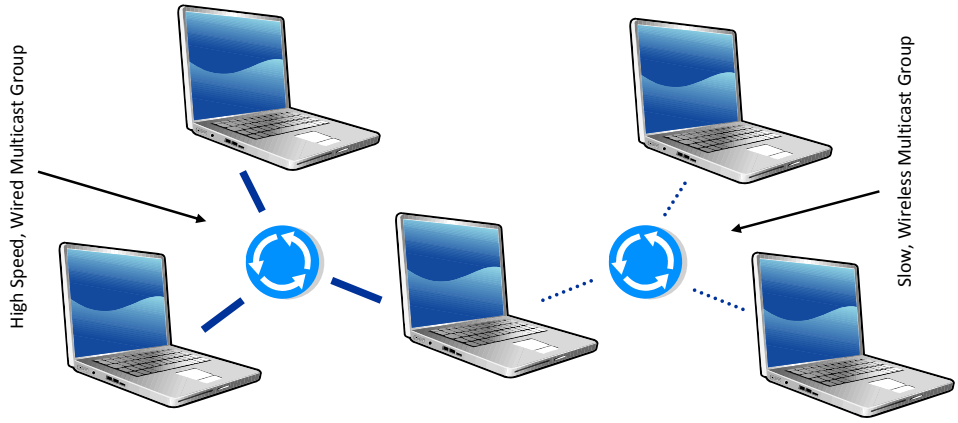


Figure 15: Reduced relay mode enabled at bridging node service to link heterogeneous network segments

Deployment to a heterogeneous network environment poses some additional challenges for Gemini. Large client publications and Gemini protocol's control messages, such as node announcements, generated by the nodes on the gigabit Ethernet segment inundate the lower bandwidth wireless segment, causing a large number of lost packets. Meanwhile, nodes in the wireless segment are reporting back with their own node announcements, which indicate the apparent loss of data from the perspective of the high bandwidth nodes. The node providing relay services is aware of the data waiting in its own transmission queue, so it can make a fairly good assessment of what is really being lost by its wireless peers. The other Ethernet nodes, though, have only the positive acknowledgements and retransmission requests to use for their assessment, and neither of these takes into account the private transmission queue used by the relay node. So they end up retransmitting quite excessive amounts of data that aren't actually needed. Also, if the gigabit Ethernet network is experiencing loss (for instance, a node joins the network late), it seems unfair that the wireless network should have to receive duplicate packets from the other multicast segment. To solve these specific problems, we implemented an optional mode called reduced relay.

The reduced relay configuration option is applied to the lower bandwidth network instances in a node service which is configured for multiple network segments, such as the center node in Figure 15. With

this configuration, the relaying node service's behavior is slightly modified. It will not relay retransmission requests, node announcements, or sequence information messages between a reduced relay network and any other network, in either direction. It will also not relay recovery (retransmission) message blocks between a reduced relay network and any other network. Simply eliminating these specific relay actions greatly improves the overall Infosphere performance for the heterogeneous deployment. Reduced relay mode causes the relay node to sort of partition the Infosphere. It's still one Infosphere – InfoObjects published on one network segment are still delivered to subscribers on the other segments. But lost packet recovery efforts do not leave a reduced network segment, and nodes on one side of a reduced relay do not discover the presence of nodes on the other side.

Reduced relay mode should also be applied to high speed network segments which are divided by a lower speed network. Such as between Local Area Network (LAN) segments which are connected via a Wide Area Network (WAN), as shown in the next figure.

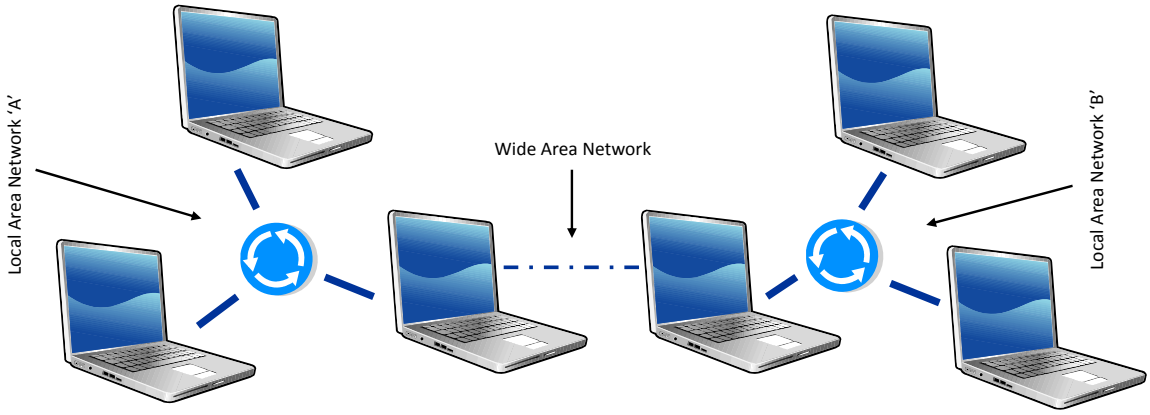


Figure 16: Reduced relay mode enabled at Node Services that relay data between LAN segments over WAN

In Figure 16, one computer from each LAN ('A' and 'B') is connected to the WAN. On these two computers, reduced relay mode should be enabled on the WAN network instance so that recovery activities which are private to LAN 'A' are not propagated to LAN 'B' and vice versa.

What happens when some blocks from a publication originating in LAN 'A' are not received by nodes in LAN 'B'? Let's assume the worst case scenario, which is that both of the relay nodes are missing some blocks from the message. In this case, the relay node in LAN 'A' will be recovered with help from other nodes in network segment 'A' through normal recovery processes, as described in 5.6.1, "Lost Information Recovery," above. Once the 'A' relay node has the complete message, it will relay the message to its other network instances, which includes the WAN. The 'B' relay node will now receive blocks from the message, but we'll assume that some of the blocks are lost in this process. So the 'B' relay will now recover blocks with help solely from the 'A' relay node. Once the 'B' relay has the full message, it will relay it to its LAN network instance. At this point, any further loss experienced by nodes on the LAN will be recovered through the normal, cooperative recovery process, but consisting of help only from machines within that network segment.

9 Gemini Network Protocol

9.1 Protocol Processing Layers

Gemini's network protocol consists of multiple layers. At the highest level, the protocol consists of messages of specific types that are shared between multiple instances of the Gemini Node Service. These messages are internally represented using regular Java objects. Message objects may be marshaled into bytes.

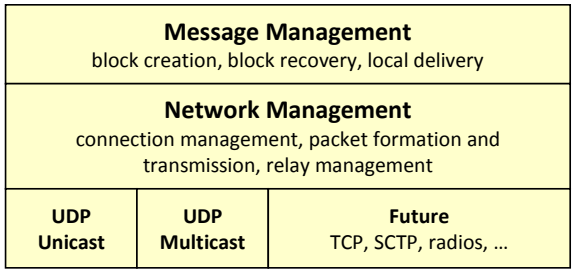


Figure 17: Node Service's Protocol Processing Layers

Messages are the domain of the Message Manager component of the node service. The Message Manager maintains the message cache, performs the bulk of the work for lost data recovery (see page 19, "5.6.1 Lost Information Recovery"), and is responsible for message delivery between other components within a single node service. The Message Manager also marshals each message into bytes and partitions the bytes into message "blocks" for transmission to a network. These blocks represent the next lower level of our protocol.

The Network Manager component receives groups of message blocks from the Message Manager. It is the network manager's responsibility to convert message blocks into datagrams for transmission. It also keeps track of the various network connections that the node service is maintaining and uses this information in support of the node service's relay capabilities.

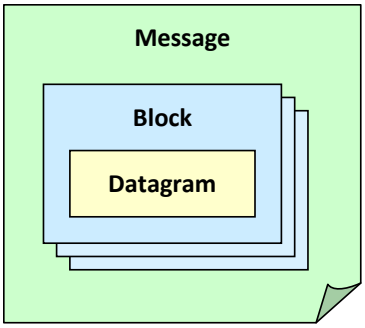


Figure 18: Messages are divided into datagrams which are represented by message blocks

In reality, the Network Manager delegates the conversion of message blocks into datagrams to the network plug-in. This datagram is the lowest level of our network protocol – it is at this point that the operating system steps in and helps us complete delivery of our data on-the-wire or over-the-air.

Figure 18 illustrates the relationship between a message and its blocks, and the one-to-one correspondence between a block and a datagram.

The lowest layer of the protocol is also pluggable. We deliver a UDP network plug-in which is capable of UDP unicast and multicast connectivity, but it is certainly feasible for one to implement a plug-in to some other transport, such as TCP, SCTP, a specific radio interface, etc.

We'll now delve into a bit more detail on the datagram or packet contents.

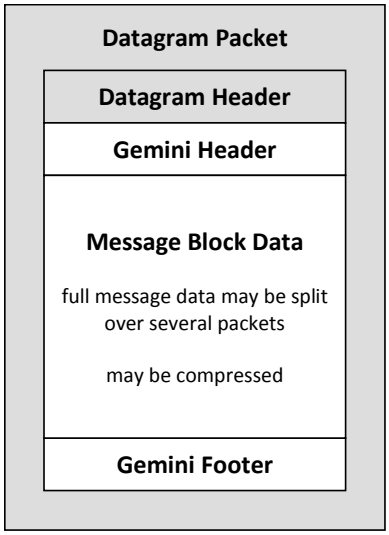


Figure 19: Gemini Datagram Parts

9.2 Datagram Representation of a Message Block

As discussed, a message is represented by some number of blocks, and each block is converted into a datagram for UDP transmission. UDP permits us to transmit datagrams (packets of partial or whole data) which are each up to 64 kilobytes in size. The size we actually use is configurable, up to this limit. So when we transmit messages larger than our configured datagram size, we also manage the fragmentation of the message while sending it, and the reassembly of the datagrams while putting the message back together.

Table 5 (page 24) lists all of the node-to-node message types. Regardless of the specific message type, the node service will generate a datagram which consists of four parts, as shown in Figure 19. UDP is responsible for population of the datagram header, which contains information such as the source and destination ports, datagram length, and checksum. We'll now describe the remaining parts in more detail.

9.2.1 Gemini Header Fields

The following table details the purpose of each of the fields of the Gemini header portion of the datagram.

Table 7: Gemini Header

Field	Purpose
Block Number	Identifies the relative location of this block for a particular multi-block message. Blocks may arrive out of order on some networks and during recovery, they do arrive out of order.
Message Hash	MD5 hash computed on the body of the entire message, prior to compression; permits detection of duplicate messages during routing and distributed queries; permits correlation between a received block and its corresponding partially reconstructed message
Total Blocks	Number of blocks generated by the Message Manager for the message to which this block is a part; permits negative acknowledgements to properly report missed block numbers; allows Message Manager to know when all blocks of a message have arrived
Delivery Mode	enumerates the delivery mode (unreliable, reliable, durable) for the corresponding message
Is Compressed	true when gzip compression has been used to reduce the size of the message's data
Type	specific type of the message for which this block is a part (see Table 5 on page 24)
Originating Node	universally unique identifier of the node service which originated this block onto the network
Hops	number of node service traversals that this block has endured so far
Maximum Hops	total number of node service traversals permitted before this block may not be relayed further
Inactivity Timeout	Partially reconstructed unreliable mode messages are held in cache for a short period of time after the most recent receipt of a datagram for the partial message, to allow for out-of-order receipt of blocks. If the timeout expires with no further blocks received, then the receiving node's Message Manager may discard the cached, partial message. A partially received reliable or durable mode message is considered "stale" when no further blocks arrive for the partial message within this period of time since the last block receipt. Once a reliable or durable partial message is stale, block recovery activities may begin.

9.2.2 Message Block Data Segment

Each message object has methods for marshaling to byte streams. This data is managed by the Message Manager component and is never interpreted by the Network Manager or its plug-ins. In fact, the Message Manager is responsible for compressing (when configured) the marshaled data, as well as dividing the data into the individual message blocks before passing them over to the network. The actual byte array stored within the message block is what is transmitted in this segment of the datagram.

9.2.3 Gemini Footer Fields

The Gemini footer records fields specific to the delivery mode of the message block being encoded. For unreliable mode messages, no footer is required. For reliable and durable mode messages, the footer contains fields which are required to correctly manage the recovery and durability of those messages.

For reliable mode messages, fields are present in the footer as detailed in Table 8.

Table 8: Gemini Footer for a Reliable Message Block

Field	Purpose
Expiration	Records the remaining number of milliseconds before the corresponding message is considered expired. A relative time is used instead of an absolute time so that the systems clocks need not be synchronized between devices running the node service. Once a message has expired, no further delivery or recovery attempts will take place and the message may be expunged from node service caches.
Serial Number	Message blocks for reliable and durable mode messages include a serial number that uniquely identifies the block among all reliable/durable blocks generated by the originating node. This is different than the message block number (in the Gemini datagram header), which is numbered sequentially from 1 for each individual message. This serial number is later reported in the form of serial number ranges which appear in Node Announcement messages and permits the positive acknowledgement based block recovery to operate. The serial number is also encoded into a special hash value used for network relay of retransmission blocks used during recovery processing.

Durable mode messages are sent reliably, so their Gemini footer data looks similar. Durable messages also require a few extra footer fields, as detailed in Table 9.

Table 9: Gemini Footer for a Durable Message Block

Field	Purpose
Expiration and Serial Number	Durable messages are also reliable; these two fields have the same purpose as is described in Table 8.
Durability Identifier	Durable messages are each assigned a universally unique identifier. This identifier is needed so that a durable message's contents may be updated over time. When the original message sender resends the durable message with new message contents, the durability identifier is used to match up (and replace) the existing durable message.
Last Durability Update	Records the time of the most recent change to the message contents associated with this message block. Nodes receiving a durable message block use this field while determining whether to ignore the block or incorporate it into a new version of the associated message.

Appendix A: Network Dissemination and Discovery Product Feature Comparison

	Relative Priority	JMS[4]	MDP	NORM	PrismTech OpenSplice DDS	RTI DDS	TAO DDS	Custom/ In-House
Bindings								
C	4	X[4]	--	--	X	X	--	x
C++	2	X[4]	X	X	X	X	X	x
Java	2	X	--	--	X	X	--	x
.NET	3	X[4]	--	--	--	--	--	x
Platforms								
Linux	2	X	X	X	X	X	X	x
Windows	2	X	X	X	X	X	X	x
Solaris	2	X	X	X	X	X	X	x
VxWorks	2	X[4]	--[3]	--[3]	X	X	X	x
Transports								
Multicast	1	X[4]	X	X	X	X	--	x
Broadcast	2	X[4]	--	--	X	--	--	x
Unicast	3	X[4]	X	X	--	X	X	x
TCP	3	X	--	--	--	--	X	x
IPv6	2	X[4]	-- ?	X		X	X	x
Pluggable Network Protocols	2	X[4]	--	--	X	X	--	x
Features [5]								
Scalability	1	--	--	--	X	X	X	x
Survivability on Dynamic Network	1	--	X	X	X	X	--	x
Delayed Delivery	2	X	X	X	X	X		x
Content Filtered Subscriptions	1	X[4]	--	--	X	X		x
Persistence	2	O	--	--	O	O	--	x
Pluggable Persistence	2		--	--	X[1]		--	x
Configurable QoS	1		--	--	X	X	X (limited)	x
Synchronous and Asynch. Msgs.	4	X	--	--	-- ?	-- ?	X (TCP)	x
Maturity	2	**High**	Low	Medium	Medium	**High**	Low	Very Low
Industry Adoption	3	**High**	Very Low	Very Low	**High**	**High**	Low	None
Performance *(Vanderbilt Tests)*	3	Low	untested	untested	High	**Highest**	Medium	untested
Major Market	2	Enterprise	File Transfer	Data Transfer	Enterprise	Embedded/ Enterprise	Research	None
Costs								
Development Licensing	2	varies	**$0**	**$0**	$36k-$48k (2-3 developers)	$45k-$90k (2-3 developers)	**$0**	$$$$$[2]
Runtime Licensing	1	varies	**$0**	**$0**	$1500/CPU	$800-$10/CPU	**$0**	**$0**

O = Optional
X = Included
-- = Not Available
? = Not 100% Certain
(empty cell) **= Unknown**

Footnotes:
1 - Persistence module available as separate purchase.
2 - Cost depends on number of features implemented.
3 - We may modify and compile the sources to new platforms.
4 - There are many vendors. Specific features vary by vendor.
5 - Terms are defined at the end of the document.

Appendix B: Acronyms and Abbreviations

B.1 Acronyms

ACE	Adaptive Communication Environment (http://www.cs.wustl.edu/~schmidt/ACE.html)
ADOCS	Automated Deep Operations Coordination System (US Army)
AFRL	Air Force Research Laboratory
API	Application Programming Interface
C2	Command and Control
C2PC	Command and Control Personal Computer (USMC)
CDRL	Contract Data Requirements List
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-the-Shelf (third party software)
CPU	Central Processing Unit
DAO	Data Access Object
DB	Database
DCGS	Distributed Common Ground System
DDS	Data Distribution Service (http://www.omg.org/docs/formal/05-12-04)
FFW	Future Force Warrior (US Army)
GIG	Global Information Grid
GNOME	GNU Object Model Environment
GNU	GNU's Not Unix
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IO	Information Object
IOR	Information Object Repository
IP	Internet Protocol (sometimes IPv4 or IPv6 to specify version 4 or 6)
IPC	Inter-process Communication
JAR	Java Archive
JBI	Joint Battlespace Infosphere
JDK	Java Development Kit
JVM	Java Virtual Machine
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
MANET	Mobile Ad-Hoc Network
MAV	Micro Air Vehicle
MD5	Message Digest 5
MTU	Maximum Transmission Unit
NAT	Network Address Translation
OIM	Operational Information Management
ORB	Object Request Broker
OS	Operating System
PC	Personal Computer
PDA	Portable Digital Assistant
PPP	Point to Point Protocol
QoS	Quality of Service
RAM	Random Access Memory
RMI	Remote Method Invocation (Java)
SAR	Synthetic Aperture Radar (imagery)
SCTP	Stream Control Transmission Protocol
SP	Service Pack (Microsoft)
SQL	Standard Query Language
SSL	Secure Sockets Layer
TAO	The ACE Orb (http://www.cs.wustl.edu/~schmidt/TAO.html)
TCP	Transport Control Protocol
TTL	Time to Live (UDP multicast)
UA	Unmanned Aircraft
UAS	Unmanned Aircraft System(s)
UGS	Unattended Ground Sensor
UDP	User Datagram Protocol
UUID	Universally Unique Identifier
WAN	Wide Area Network
XML	Extensible Markup Language

B.2 Abbreviations

CAPI	Java Common Client API
CONUS	Continental United States
InfoObject	Information Object; a piece of data with a standardized header (type, version, etc.) and optional XML and binary data segments
Infosphere	Information space; one's group of information sharing peers (http://www.infospherics.org/)
MB	Megabyte (1024^2 bytes)
netem	Network Emulation as part of Linux kernel 2.6